Research Reports in Software Engineering and Management 2008:02

Proceedings of the 3rd Workshop on Quality in Modeling

Jean-Louis Sourrouille Miroslaw Staron (Eds.)

IT University of Göteborg

CHALMERS | GÖTEBORGS UNIVERSITET

Department of Applied IT

Research reports in Software Engineering and Management
Report number 2008:02
Series editor: Lars Pareto

ISSN: 1654-4870

Department of Applied Information Technology

IT University of Göteborg

Göteborg University and Chalmers University of Technology

PO Box 8718

SE – 402 75 Göteborg

Sweden

Telephone + 46 (0)31-772 4895

Proceedings of the 3rd

Workshop on Quality in Modeling

Co-located with

MoDELS 2008

The ACM / IEEE 11th International Conference on

Model Driven Engineering Languages and Systems

Editors

Jean-Louis Sourrouille
Miroslaw Staron

Organizers

Jean-Louis Sourrouille, chair,
INSA Lyon, France

Miroslaw Staron, program chair,
IT University of Göteborg, Sweden

Ludwik Kuzniarz,
Blekinge Institute of Technology, Ronneby, Sweden

Parastoo Mohagheghi,
SINTEF ICT, Norway

Lars Pareto,
IT University of Göteborg, Sweden

Program committee

Preface

Quality is an important issue in software engineering, and the stakeholders involved in the development of software systems definitely are aware of the impact of the quality of both development process and produced artifacts on final software product quality. The recent introduction of Model Driven Software Development (MDD) raises new challenges related to ensuring proper quality of the software produced when using this approach. Software quality management within MDD is widely researched from multiple new perspectives. Furthermore, in software engineering, the issues of model quality need to be approached from the viewpoints of both industry practices and academic research in order to arrive at sound and industrially applicable results.

This workshop is built upon the experience and discussions during the previous workshops on Quality in Modeling. It aims to gather researchers and practitioners interested in the emerging issues of quality in the context of MDD, and to provide a forum for presenting and discussing emerging issues related to software quality in MDD. The intended result is to increase consensus in understanding quality of models and issues that influence the quality.

The intention of this year's workshop is to devote a part of the discussion to model quality related to model driven software development processes. Within "usual" software development, software process quality and project management quality are widely used, while code quality seems to be an under-exploited way of improving software quality. However, all the concepts and theory about code quality have been widely described. Therefore, a special attention is to be paid to practical issues such as the introduction of model quality into the software development process in a convenient and accepted way.

The workshop is divided in two parts:

- Presentation part: presentation and discussion of the contributions of the accepted papers,
- Working part: guided discussion based on a presentation by an industrial practitioner and questions sent to the participants, followed by discussing a road map for further research.

The presentation part consists of two sessions for the presentation of accepted papers:

- Towards model quality,
- Frameworks for model quality.

The working part is also divided in two sessions: introducing model quality and ideas for future research. The rationale behind the first session of the working part is to carry out a discussion about the introduction of model quality into software development process by drawing a parallel with the management of code quality. First, an industrial practitioner will introduce practical aspects regarding code quality in actual software development. Then, based on a list of prepared questions, participants will discuss the practical solutions adopted for code quality and their

suitability for model quality. The emergence of pending issues is expected from the discussion. The second session of the working part will deal with desirable future works and research interests of the participants, aiming to draw a map of the promising research directions for Quality in Modeling.

The summary and results of the working sessions will be published in the post-workshop report.

Workshop organizers

Table of contents

Design of a Functional Size Measurement Procedure for a Model-Driven Software Development Method*

Beatriz Marín[1], Nelly Condori-Fernández[1] and Oscar Pastor[1]

[1] Centro de Investigación en Métodos de Producción de Software,
Universidad Politécnica de Valencia,
Camino de Vera s/n,
46022 Valencia, Spain
{bmarin, nelly, opastor}@pros.upv.es

Abstract. The capability to accurately quantify the size of software developed with a Model-Driven Development (MDD) method is critical to software project managers for evaluating risks, developing project estimates, and having early project indicators. This paper presents a measurement procedure defined according to the last version of the ISO 19761 standard measurement method. The measurement procedure has been designed to measure the functional size of object-oriented applications generated from their conceptual models by means of model transformations. The measurement procedure is structured in three phases: the strategy phase, where the purpose of the measurement is defined; the mapping phase, where the elements of the conceptual model that contribute to the functional size are selected; and the measurement phase, where the functional size of the generated application is obtained.

Keywords: Conceptual model, Object orientation, Functional size measurement, COSMIC, MDD.

1 Introduction

Models are abstractions of the reality that help to understand complex problems and their potential solutions [22]. Model-Driven Development (MDD) methods have been developed to take advantage of the benefits of the use of models: a simplified view of the problem (using concepts that are much less bound to the underlying implementation technology and are much closer to the problem domain); and an easy way to specify, understand, and maintain the model. Since MDD methods are focused on models and model transformations, these allow the achievement of the automatic generation of the final product. To do this, the models (conceptual models) must have enough semantic formalization to specify all the functionality of the final application and also to avoid different interpretations for the same model.

* This work has been developed with the support of MEC under the project SESAMO TIN2007-62894 and co financed by FEDER.

The adoption of MDD methods has presented new challenges, such as the need to accurately quantify the functional size of the generated products from their conceptual models. Since the functional size of applications is essential to apply estimation models, defect models, and budget models [17], it is very important to obtain the functional size of the applications so that the project leader generates indicators to facilitate the project management and to assure the quality of the final product.

To measure the functional size of software applications, four measurement methods have been recognized as standards: IFPUG FPA [13], MK II FPA [14], NESMA FPA [15], and COSMIC FFP [12]. These methods have been illustrated in the measurement of the functional size of final applications. However, project leaders need indicators in the early stages of software development for a better management of MDD projects. For this reason, it is necessary to define how the measurement standards can be applied to the conceptual models that allow the generation of the final application. The specification of the way in which the measurement method must be applied to a phase of the development of a software application is named measurement procedure [9].

The COSMIC measurement method can be applied to any type of software and allows the measurement of multi-layer applications, in contrast to other functional size measurement methods (such as IFPUG FPA, NESMA FPA, and MK II FPA). For this reason, we have selected the COSMIC measurement method to specify a measurement procedure that can be applied to conceptual models.

The objective of this work is to design a measurement procedure that allows the application of the COSMIC measurement method to conceptual models, which are used by a MDD method to generate a final application by means of model transformations. Thus, the project leader will have the accurate functional size available to calculate productivity indicators, the price to be charged to clients, the defects in the models, etc.

The rest of the paper is organized as follows: Section 2 presents the phases and activities of the last version of the COSMIC measurement method and the measurement procedures based on COSMIC to measure conceptual models. Section 3 presents the design of a measurement procedure that applies COSMIC to measure the functional size of final applications from their object-oriented conceptual models. Finally, Section 4 presents some conclusions and further work.

2 Background and Related Works

The ISO/IEC 14143-1 [11] standard defines *functional size* as the size of the software derived by quantifying the functional user requirements. This standard also defines a Functional Size Measurement (FSM) as the process of measuring the functional size. In addition, this standard defines a FSM method as the implementation of a FSM that is defined by a set of rules, which is defined in accordance with the mandatory features defined in the ISO/IEC 14143-1.

The COSMIC measurement method was first recognized as a standard measurement method [12] because it fulfilled the characteristics defined in the ISO/IEC 14143-1 [10] and was verified with the ISO/IEC 14143-2 [11]. Later, the

COSMIC measurement method was improved maintaining the concepts and the characteristics that allowed it to be recognized as a standard method. The last version of the COSMIC measurement method is version 3.0 [1], which is different from the previous version mainly because it has a new phase to define the measurement strategy and it changes the concepts of end-user viewpoint and developer viewpoint for a generic concept named functional user, which allows the measurement of each piece of software that makes up an application. Next, we describe in more detail this version of the COSMIC measurement method.

2.1 The COSMIC Measurement Method

The application of the COSMIC measurement method [1] includes three phases: the measurement strategy, the mapping of concepts, and the measurements of the identified concepts (see Figure 1).

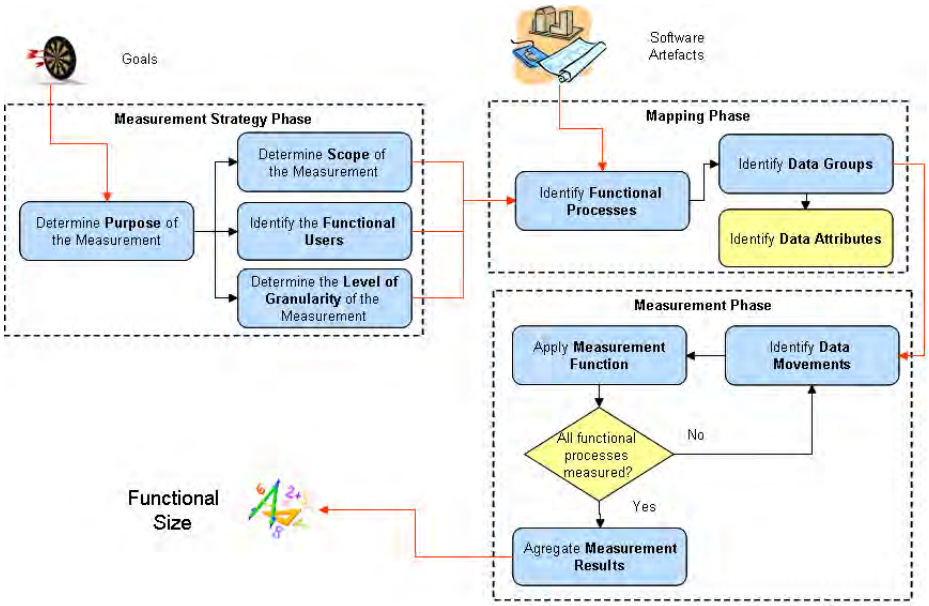


Fig. 1. Phases and activities in the COSMIC measurement method.

In the *measurement strategy phase,* the purpose of the measurement exercise must be defined to explain why it is necessary to measure and what the measurement result will be used for. Next, the scope of the measurement must be defined in order to allow the set of user functional requirements that will be included in the measurement task to be selected. Then, the functional users of the application to be measured must be identified. The functional users are the types of users that send (or receive) data to (from) the functional processes of a piece of software. This phase also includes the identification of the boundary, which is a conceptual interface between the functional

user and the piece of software that will be measured. Finally, the level of granularity of the description of a piece of software to be measured is identified.

In the *mapping phase,* the functional processes must be identified (i.e., the elementary components of a set of functional user requirements). Every functional process is triggered by a data movement from the functional user, and the functional process is completed when it has executed all the data movements required for the triggering event. It should be kept in mind that a triggering event is an event that causes a functional user of the piece of software to initiate one or more functional processes. Next, the data groups must be identified. This is a set of data attributes that are distinct, non empty, non ordered, non redundant, and that participates in a functional process. Finally, the identification of the data attributes, which comprise the smallest part of information of a data group, is optional.

In the *measurement phase,* the data movements (Entry, Exit, Read and Write) for every functional process must be identified. When all the data movements of the functional process are identified, the measurement function for the functional process must be applied. This is a mathematical function that assigns 1 CFP (Cosmic Function Point) to each data movement of the functional process. Then, after all the functional processes are measured, the measurement results are aggregated to obtain the functional size of the piece of software that has been measured.

2.2 Related Works

There are some approaches that apply COSMIC (in any of its versions) in order to estimate the functional size of future software applications from the conceptual model specifications [3] [5] [8] [16]. These proposals use scenarios, use case diagrams, sequence diagrams, and i* models to estimate the functional size. Therefore, these proposals estimate the functional size in conceptual models that not are used to generate the final application because these models do not have enough semantic expressiveness to specify all the functionality (for instance, in these models is not possible to specify the way in that the values of the attributes of a class change). Therefore, the functional size obtained by these proposals is not the accurate functional size of the final application. For this reason, the project leader can not use the functional size obtained to calculate indicators or use quality models (budget models, defect models, etc).

To avoid these problems, other proposals have been designed to measure the functional size of conceptual models that have more expressiveness to specify the functionality of the final applications and that allow the automatic generation of the final applications from these models. This is the case of Diab's proposal [7] and Poels' proposal [20]. Diab's proposal presents a measurement procedure to measure real time applications modelled with the ROOM language [23]. Diab's proposal uses a kind of statechart diagrams to measure the functional size. Poels' proposal presents a measurement procedure to object-oriented applications modelled with an event-based method named MERODE [6]. Poels' proposal allows the measurement of the functional size of Management Information Systems (MIS). The main disadvantage of both proposals is that the conceptual model does not allow the specification of all the functionality of the final application; for instance, that conceptual model does not

allow the specification of the presentation of the application. Also, Poels' proposal is restricted to a specific technology because it uses the AndroMDA tool to specify the presentation of the application and to generate the final application. In addition, both proposals were defined using an old version of the COSMIC measurement method, and, therefore, these proposals do not take into account the improvements made to the COSMIC measurement method, for instance, the capability to measure the functional size of a piece of software of the application depending on the functionality that needs other piece of software.

None of the proposals for measurement procedures based on COSMIC allows the measurement of the accurately functional size of MIS applications in the conceptual model. Moreover, none of them take into account the improved version of COSMIC. The main limitation of the approaches presented above comes from the lack of expressiveness of the conceptual model that allows the generation of the final application. If the conceptual model has enough expressiveness to specify all the functionality of the final application, then a measurement procedure can accurately measure the functional size of the final application from its conceptual model.

The OO-Method approach [18] is an object-oriented method that allows the automatic generation of final applications by means of model transformations. It provides the semantic formalization needed to define complete and unambiguous conceptual models, allowing the specification of all the functionality of the final application in the conceptual model. This method has been implemented in a tool [4] that allows the automatic generation of fully working applications. The applications generated can be desktop or web MIS applications and can be generated in several technologies (for instance, java, C#, visual basic, etc.). The measurement procedure presented in this paper is based on this MDD method.

3 A FSM Procedure for Conceptual Models of an MDD Method

The design of a measurement procedure is a key stage in the development of a measurement procedure because the objective of the measurement, the artifact that will be measured, the measurement rules, and the measurement strategy are defined in this stage. It is very important to correctly perform the design of a procedure of measurement (correctly abstracting the elements that will be measured), since, otherwise, the procedure may not measure what should be measured according to the specifications in the base measurement method selected. It is also important to keep in mind the direct influence that the design of a measurement procedure has on the application of this procedure. For instance, if the design is incorrect, then the application of the procedure may be confused and erroneous measures may be obtained.

Since design is very important, in this section of the paper we present the design of a measurement procedure. In the design, the last version of the COSMIC measurement method has been selected. Therefore, the measurement procedure has three phases: strategy, mapping, and measurement. Moreover, the measurement procedure has been designed in the context of the OO-Method conceptual model because this model has the expressivity necessary to specify all the functionality of

the final application. The following present the phases and the activities of the COSMIC method instantiated with concepts of the OO-Method conceptual model.

3.1 The Measurement Strategy Phase

Initially, the purpose of the measurement must be determined. The scope and the granularity level are determined depending on the purpose. Finally, the functional users are identified.

Purpose. The purpose of the measurement procedure has been defined in terms of a Goal-Question-Metric template [2]. Therefore, the purpose is:
To define a measurement procedure
with the purpose of applying the COSMIC measurement method to applications generated in an MDD environment
with respect to its functional size
from the point of view of the researcher
in the context of the conceptual model of an MDD development process named OO-Method.

Scope. The scope of the measurement procedure is the conceptual model of the OO-Method MDD technology. This conceptual model is comprised of four models: the object model, the dynamic model, the functional model, and the presentation model. The object model defines the structure and static relationships between the classes. The dynamic model defines the possible valid lives for the objects of a class and the interaction among objects. The functional model captures the semantics associated to object state changes, triggered by the occurrence of events. Finally, the presentation model allows the specification of the user interfaces in an abstract way. With all of these models, the conceptual model has all the details needed for the generation of the final application. The complete definition of the elements of the conceptual model of OO-Method is described in detail in [19].

Since the OO-Method software applications are generated according to a three-tier software architecture that is structured in a hierarchy, we distinguish three independent *layers* of the final application: the client layer, the server layer, and the database layer. Also, we distinguish three *pieces of software* that correspond to the parts of the application in each layer (see Figure 2).

Granularity Level. Since the conceptual models need the functional requirements to be detailed and validated to generate the final application, the granularity level is low.

Functional Users. The functional users in the final applications are: (1) the human users of the application, and (2) the pieces of software that interchange data between the layers of the application. The functional users are separated by a boundary from the pieces of software of the application.

The functional users can be specified in the conceptual model by the role that the user has been assigned in order to execute the services of the application. In the OO-

Method approach, the different roles of the users are specified in the object model as agents of the services of classes that can execute. These users are functional users of the client piece of software of the application because they send (or receive) data to (from) this piece of software (see Figure 2).

On the other hand, the functional users that correspond to the pieces of software of a three-tier application are the client piece of software and the server piece of software (see Figure 2). The client piece of software is a functional user of the server piece of software because it interchanges data with this piece of the application. The server piece of software is a functional user of the client piece and the database piece of software because it sends (or receives) data to (from) these pieces of software.

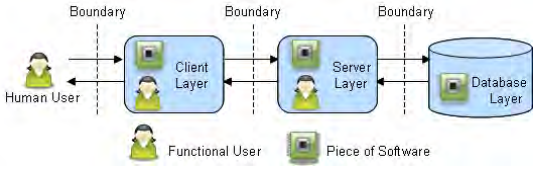


Fig. 2. Functional users and scope of an OO-Method application.

To avoid mistakes in the identification of the functional users and the boundaries of an OO-Method application, Table 1 shows three rules that have been defined to identify the functional users (Rule 1, 2, and 3) and one rule that has been defined to identify the boundaries (Rule 4).

3.2 The Mapping Phase

In this phase, the functional processes must be identified, and after that, the data groups and the data attributes must be identified.

Functional Process. A functional process is a set of functionalities of the application that allows the achievement of a functional requirement. Generally, in the final application, the functional requirements are presented in groups of functionality that can be directly accessed from the graphical user interface (GUI), for instance, in the menu options. Since the MDD conceptual models can specify the presentation of the final application, the groups of functionality (or interaction units) that can be directly accessed in the GUI of the final application are considered to be a functional process (see Rule 5 in Table 1).

It is important to note that all the functionalities that can be accessed or executed from the interaction units make up the functional process and not just the interaction unit that can be accessed directly from the menu of the application. Therefore, once all the elements that make up the interaction unit are identified, the functional process is correctly identified.

The interaction units can be used to (1) show information to the user or (2) to execute services by the user. The interaction units that show information can show data of an object or data of a set of objects. To do this, these interaction units basically use presentation patterns to display information of the objects, to filter the objects,

and to access other interaction units. On the other hand, the service interaction unit uses presentation patterns to enter the arguments of the service and to access other interaction units (for instance, to search for an object that corresponds to an argument of the service). To completely identify the elements that make up a functional process, the following rules must iteratively apply:

Rule 5.a: Identify the **display pattern**, the **filter pattern** and the **interaction units** that can be accessed from the functional process as elements of the functional process.

Rule 5.b: Identify the **arguments** and the **interaction units** that can be accessed from the functional process as elements of the functional process.

With the rules described above, the functional processes can be identified several times if they are accessed from more than one access in the menu of the final application. Also, the interaction units that are elements of a functional process can be accessed by several components of the functional process. To avoid duplicity in the identification of the functional processes and the elements that compose it, the following rules have been defined:

Rule 5.c: Drop the interaction units contained in a functional process when these interaction units also correspond to a functional process.

Rule 5.b: Identify the interaction units contained in a functional process only once.

Data Group. The data groups are the conceptual objects of an application. In object oriented applications, the data groups correspond to the classes of the object model. However, if the class participates in an inheritance hierarchy, one data group must be identified for the father of the hierarchy, and one data group must be identified for each child that has attributes different from its father. Rules 6, 7, and 8 of Table 1 have been defined for the correct identification of the data groups.

Attributes. The attributes correspond to the attributes of the classes specified in the object model, which have been identified as data groups (see Rule 9 of Table 1).

Table 1. Mapping Rules.

COSMIC	OO-Method
Functional User	Rule 1: Identify 1 functional user for each **agent** in the OO-Method object model. Rule 2: Identify the client functional user for the **server piece of software** of an OO-Method application. Rule 3: Identify the server functional user for the **client piece of software** of an OO-Method application.
Boundary	Rule 4: Identify 1 boundary **between** a **functional user** and a **piece of software** of an OO-Method application.
Functional Process	Rule 5: Identify 1 functional process for each **interaction unit** that can be **directly** accessed in the menu of the OO-Method presentation model.
Data Group	Rule 6: Identify 1 data group for each **class** defined in the OO-Method object model, which does not participate in an inheritance hierarchy. Rule 7: Identify 1 data group for each **parent class** of an inheritance hierarchy defined in the OO-Method object model. Rule 8: Identify 1 data group for each **child class** of an inheritance hierarchy of the OO-Method object model, which has different attributes from its father.

Attributes	Rule 9: Identify the set attributes of the **classes** defined in the OO-Method object model.

3.3 The Measurement Phase

In this phase, the data movements of each functional process are identified. Then, a measurement function is applied, and the results are aggregated to obtain the functional size of each functional process. Finally, the functional sizes of the functional processes are aggregated to obtain the functional size of the piece of software that has been measured.

Data Movements. Each functional process has two or more data movements. Each data movement moves a single data group. A data movement can be an *Entry* (E), an *Exit* (X), a *Read* (R), or a *Write* (W) data movement.

An *Entry data movement* is a data movement that crosses the boundary from a functional user to a functional process. An *Exit data movement* is a data movement that crosses the boundary from a functional process to a functional user. A *Read data movement* is a data movement that crosses the boundary from the database to a functional process. Finally, a *Write data movement* is a data movement that crosses the boundary from a functional process to the database of the application. The data movements that can occur in an OO-Method application are shown in Figure 3.

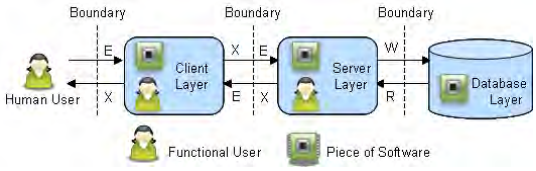


Fig. 3. Data movements that can occur in an OO-Method application.

To identify the data movements that occur in an OO-Method application, 29 rules were defined. These rules are grouped by the conceptual elements of the model. Each rule considers the type of the data movement, the piece of software, and the element of the OO-Method conceptual model.

In the identification of the functional processes, the smallest elements contained in the functional processes were identified as display patterns, filter patterns, and services. With the specification of these patterns in the conceptual model, it is possible to identify all the data movements that can occur in the final application.

The *display patterns* must be identified in the presentation model of the application. The display patterns define the attributes of classes of the object model that will be shown to the users of the application (human functional users). Rules 10, 11, 12, and 13 of Table 2 have been defined to identify the data movements of the display patterns of an application.

The *filter patterns* must also be defined in the presentation model of the application. The filter patterns specify the data that will be shown to the user, if a

formula calculated with certain input variables has a particular value. These input variables must be entered by the users of the application (human functional users), and the application retrieves a set of data that satisfies the filter formula calculated with the values of the input variables. The filter patterns are defined in the context of a class of the object model and can retrieve information about this class and the classes related by association with this class. Rules 14, 15, 16, 17, and 18 of Table 2 have been defined to identify the data movements of the display patterns of an application.

The *services* are defined in the classes of the object model. The services have a set of inbound arguments that allows the execution of the service itself. This execution can be changes in the values of the attributes of the class that contains the service, creation of associations between classes, execution of a set of services of a class, execution of a set of services of the model, etc. Therefore, the services defined in the OO-Method object model can be classified as event, transaction/operation, and global services.

The events can be defined to change the value of the attributes of a class. To do this, the events use formulas called valuations (see Rules 19, 20, 21 of Table 2). Also, the events can be defined to create instances of a class or to destroy instances of a class by means of a property of the events. Finally, some events can be defined to create or destroy associations between classes (see Rule 21 of Table 2).

The transactions and operations are defined in a class to group a set of services that must be sequentially executed. These services can belong to the class that contains the transaction or operation. These services can also belong to the classes associated with the class that contains the definition of the transaction or operation. The global services are defined in the OO-Method object model in order to group a set of services of different classes, which may or may not be associated. These kinds of services have been considered in Rule 22 of Table 2.

The arguments of each service are defined in the object model, and a single default value for the arguments of the service can be specified. In addition, a formula can be specified to initialize the values of the arguments of a service. Occasionally, some arguments of a service can depend on the value of other arguments of the service. To represent this situation, dependency rules are used in the OO-Method conceptual model. To count the functionality related to the arguments of a service, Rules 23, 24, 25, 26, 27, 28, 29 and 30 of Table 2 have been defined.

Before the execution of a service, the preconditions of the service must be checked. If the preconditions are satisfied, it is possible to continue with the execution of the service. Otherwise, an error message is shown to the user of the application (human functional user). Rules 31, 32, and 33 of Table 2 have been defined to take into account the functionality of the preconditions of a service.

On the other hand, after the execution of a service, the integrity constrains of the class that contains the service are checked. If the constraints are satisfied, the service ends its execution. If not, the service performs a rollback of the execution and shows a message to the user of the application (human functional user). Rules 34, 35, and 36 of Table 2 have been defined to take into account the functionality of the integrity constraints of the class that contains the service.

Finally, the conditions (guards) that must be fulfilled to execute a service that changes the state of an object (see Rule 37 of Table 2) and the conditions that must be

fulfilled to trigger a service (see Rule 38 of Table 2) can be specified in the dynamic model.

Table 2. Rules to identify the data movements of an OO-Method application.

Conceptual Element	Rules
Display Pattern	Rule 10: Identify **1X** data movement for the client piece of software for each **display pattern** in the interaction units that participate in a functional process. Rule 11: Identify **1E** data movement for the client piece of software, and **1X** and **1R** data movements for the server piece of software for each different **class** that contributes with attributes to the display pattern. Rule 12: Identify **1R** data movement for the server piece of software for each different **class** that is used in the **condition** of the **derivation formula** of derivate attributes that appear in the display pattern. Rule 13: Identify **1R** data movement for the server piece of software for each different **class** that is used in the **effect** of the **derivation formula** of derivate attributes that appear in the display pattern.
Filter Pattern	Rule 14: Identify **1E** data movement and **1X** data movement for the client piece of software, and **1E** data movement for the server piece of software for the **set of data-valued variables** of the filter patterns (represented by the class that contains the filter) of the interaction units contained in a functional process. Rule 15: Identify **1E** data movement and **1X** data movement for the client piece of software, and **1E** data movement for the server piece of software for each different **object-valued variable** of the filter patterns of the interaction units contained in a functional process. Rule 16: Identify **1R** data movement for the server piece of software for each different **class** that is used in the **filter formula** of the filter patterns of the interaction units that participate in a functional process. Rule 17: Identify **1E** data movement and **1X** data movement for the client piece of software, and **1X** data movement for the server piece of software for the **set of data-valued variables** with a **default value** of the filter patterns (represented by the class that contains the filter) of the interaction units contained in a functional process. Rule 18: Identify **1E** data movement and **1X** data movement for the client piece of software, and **1X** data movement for the server piece of software for each different **object-valued variable** with **default value** of the filter patterns of the interaction units contained in a functional process.
Service	Rule 19: Identify **1R** data movement for the server piece of software for each different **class** that is used in the **condition** of the **valuation formula** of events that participate in the interaction units contained in a functional process. Rule 20: Identify **1R** data movement for the server piece of software for each different **class** that is used in the **effect** of the **valuation formula** of events that participate in the interaction units contained in a functional process. Rule 21: Identify **1W** data movement for the server piece of software for each **create event**, **destroy event**, or **event that has valuations** (represented by the class that contains the service) that participate in the interaction units contained in a functional process. Rule 22: Identify **1R** data movement for the server piece of software for each different **class** that is used in the **service formula** of transactions, operations, or global services that participate in the interaction units contained in a functional process.

	Rule 23: Identify **1E** data movement and **1X** data movement for the client piece of software, and **1E** data movement for the server piece of software for the **set of data-valued arguments** of the services (represented by the class that contains the service) that participate in the interaction units contained in a functional process. Rule 24: Identify **1E** data movement and **1X** data movement for the client piece of software, and **1E** data movement for the server piece of software for each different **object-valued argument** of the services that participate in the interaction units contained in a functional process. Rule 25: Identify **1E** data movement and **1X** data movement for the client piece of software, and **1X** data movement for the server piece of software for the **set of data-valued arguments** with a **default value** of the services (represented by the class that contains the service) that participate in the interaction units contained in a functional process. Rule 26: Identify **1E** data movement and **1X** data movement for the client piece of software, and **1X** data movement for the server piece of software for each different **object-valued argument** with a **default value** of the services that participate in the interaction units contained in a functional process. Rule 27: Identify **1R** data movement for the server piece of software for each different **class** that is used in the **condition** of the **initialization formula** of the arguments of the services that participate in the interaction units contained in a functional process. Rule 28: Identify **1R** data movement for the server piece of software for each different **class** that is used in the **initialization formula** of the arguments of the services that participate in the interaction units contained in a functional process. Rule 29: Identify **1R** data movement for the server piece of software for each different **class** that is used in the **condition** of the **dependency formula** of the services that participate in the interaction units contained in a functional process. Rule 30: Identify **1R** data movement for the server piece of software for each different **class** that is used in the **dependency formula** of the services that participate in the interaction units contained in a functional process. Rule 31: Identify **1R** data movement for the server piece of software for each different **class** that is used in the **precondition formulas** of the services that participate in the interaction units contained in a functional process. Rule 32: Identify **1X** data movement for the client piece of software for all **error messages** of the **precondition formulas** of the services that participate in the interaction units contained in a functional process. Rule 33: Identify **1E** data movement for the client piece of software, and **1X** data movement and **1R** data movement for the server piece of software for each different **class** used in the **error messages** of the **precondition formulas** of the services that participate in the interaction units contained in a functional process. Rule 34: Identify **1R** data movement for the server piece of software for each different **class** that is used in the **integrity constraint formulas** of the class that contains each service that participates in the interaction units contained in a functional process. Rule 35: Identify **1X** data movement for the client piece of software for all **error messages** of the **integrity constraint formula** of the class that contains each service that participates in the interaction units contained in a functional process. Rule 36: Identify **1E** data movement for the client piece of software, and **1X** data movement and **1R** data movement for the server piece of software for each different **class** used in the **error messages** of the **integrity constraint formula** of the class that contains each service that participates in the interaction units contained in a functional process.

	Rule 37: Identify **1R** data movement for the server piece of software for each different **class** that is used in the **condition formula** of a **transition** that changes the state of an object by means of a service that participates in the interaction units contained in a functional process. Rule 38: Identify **1R** data movement for the server piece of software for each different **class** that is used in the **trigger formula** that triggers a service that participates in the interaction units contained in a functional process.

Measurement Function. The measurement function assigns 1 CFP (Cosmic Function Point) to each data movement that occurs in a functional process of the application.

Measurement Aggregation. Once the measurement function has been applied, the measures can be aggregated to obtain the functional size of each functional process of each piece of software of the application as well as the whole application. Since the functional size of each functional process corresponds to the addition of the data movements that occur in this functional process, the data movements are aggregated to obtain the functional size of each functional process. Using the same criteria it is possible to obtain the functional size of each piece of software. Therefore, the functional size of each piece of software corresponds to the addition of the data movements that occur in the functional processes that are contained in this piece of software. In the case of the functional size of the whole application, the same criteria have been used. Therefore, the functional size of the whole application corresponds to the addition of the data movements that occur in the functional processes that are contained in the pieces of software that are contained in the application. Table 3 presents the rules defined to obtain the functional size.

Table 3. Rules to obtain the functional size of the functional processes, the pieces of software of the application, and the whole application.

Conceptual Level	Rules
Functional Process	Rule 39: Aggregate the CFP related to the **data movements** identified in the **client piece** of software of each **functional process** to obtain the functional size of that process. Rule 40: Aggregate the CFP related to the **data movements** identified in the **server piece** of software of each **functional process** to obtain the functional size of that process.
Piece of Software	Rule 41: Aggregate the CFP related to the **data movements** identified in the **functional processes** identified in the **client piece** of software to obtain the functional size of that piece of software. Rule 42: Aggregate the CFP related to the **data movements** identified in the **functional processes** identified in the **server piece** of software to obtain the functional size of that piece of software.
Application	Rule 43: Aggregate the CFP related to the **data movements** identified in the **functional process** contained in the **pieces of software** identified in the **application** to obtain the functional size of the whole application.

Finally, with the rules of the measurement procedure, it is possible to accurately measure the functional size of the OO-Method software applications that are generated from their conceptual models. The conformity of the rules that are defined in this measurement procedure with the COSMIC measurement method has been validated by experts. In addition, this measurement procedure has been applied to OO-Method case studies, and the results have been compared with the measures obtained by experts. In terms of theoretical validation, since the validation of COSMIC has been carried out successfully from the perspective of measurement theory in [5] using the DISTANCE framework [21], and since the measurement procedure has been designed on the basis of COSMIC, we can infer that the measurement procedure has also been theoretically validated.

4 Conclusions and Further Work

In this paper, we have presented a measurement procedure, which is an FSM procedure for applications that are generated from object-oriented conceptual models of an MDD method. This procedure was designed in accordance with the COSMIC measurement method, which facilitates the functional size measurement of multi-layer applications (in contrast to traditional FSM methods).

The measurement procedure has been designed to obtain accurate measures of applications that have been generated from their conceptual models. It is important to note that it is possible to obtain the accurate functional size because all the functionality of the final application has been specified in the conceptual model, which is automatically transformed to the final application. In other cases (i.e., the conceptual model is not automatically transformed to the final application), it is only possible to obtain estimations of the functional size. Assuming that the conceptual model is of high quality (that is, the conceptual model is correct, complete, and without defects), the measurement procedure could be completely automated, providing measurement results in a few minutes using minimal resources. Obviously, if the conceptual model has incorrect information or missing information, the measures obtained will not be correct by any measurement procedure.

This paper defines a set of mapping rules that allow the selection of the relevant conceptual elements of a specific MDD method called OO-Method to measure the functional size according to the COSMIC concepts. Moreover, a set of measurement rules has been defined to obtain the functional size at three levels: the functional process level, the piece of software level, and the whole application level. The mapping and measurement rules were defined in the context of OO-Method, but many conceptual constructs of the OO-Method conceptual model can be found in other object-oriented methods. For this reason, the measurement procedure could be generalized to other object-oriented MDD methods.

The main limitation of the measurement procedure presented in this paper is the large amount of time required for the manual application of the procedure to models of real applications. We plan to develop a tool that automates the measurement procedure and to conduct empirical studies of the tool to ensure the accuracy of the measures.

References

1. Abran, A., Desharnais, J., Oligny, S., St-Pierre, D., Symons, C.: The COSMIC Functional Size Measurement Method, version 3.0. In: GELOG web site www.gelog.etsmtl.ca (2007)
2. Basili, V., Rombach, H.: The TAME Project: Towards Improvement Oriented Software Environments. IEEE Transactions on Software Engineering, 758--773 (1988)
3. Bévo, V., Lévesque, G., Abran, A.: Application de la méthode FFP à partir d'une spécification selon la notation UML: compte rendu des premiers essais d'application et questions. In: 9th IWSM, Lac Supérieur, Canada, pp. 230--242 (1999)
4. CARE Technologies Web Site, www.care-t.com
5. Condori-Fernández, N.: Un procedimiento de medición de tamaño funcional a partir de especificaciones de requisitos. Doctoral thesis, Univ. Politécnica de Valencia, Spain (2007)
6. Dedene, G., Snoeck, M.: M.E.R.O.DE.: A Model-driven Entity-Relationship Object-oriented Development Method. ACM SIGSOFT Software Engineering Notes 19(3), 51--61 (1994)
7. Diab, H., Frappier, M., St-Denis, R.: Formalizing COSMIC-FFP Using ROOM. In: ACS/IEEE Int. Conf. on Computer Systems and Applications (AICCSA), pp. 312 (2001)
8. Grau, G., Franch, X.: Using the PRiM method to Evaluate Requirements Model with COSMIC-FFP. In: International Conference on Software Process and Product Measurement (IWSM-MENSURA), Mallorca, Spain (2007)
9. ISO, International vocabulary of basic and general terms in metrology (VIM), Geneva, Switzerland (2004)
10. ISO, ISO/IEC 14143-1 – Information Technology – Software Measurement – Functional Size Measurement – Part 1: Definition of Concepts (1998)
11. ISO, ISO/IEC 14143-2 – Information Technology – Software Measurement – Functional Size Measurement – Part 2: Conformity Evaluation of Software Size Measurement Methods to ISO/IEC 14143-1:1998 (2002)
12. ISO/IEC 19761, Software Engineering – CFF – A Functional Size Measurement Method (2003)
13. ISO/IEC 20926, Software Engineering – IFPUG 4.1 Unadjusted Functional Size Measurement Method – Counting Practices Manual (2003).
14. ISO/IEC 20968, Software Engineering – Mk II Function Point Analysis – Counting Practices Manual (2002)
15. ISO/IEC 24570, Software Engineering – NESMA Functional Size Measurement Method version 2.1 – Definitions and Counting Guidelines for the application of Function Point Analysis (2005)
16. Jenner, M.S.: COSMIC-FFP and UML: Estimation of the Size of a System Specified in UML – Problems of Granularity. In: 4th European Conf. Soft. Measurement and ICT Control, pp. 173--184 (2001)
17. Meli, R., Abran, A., Ho Vinh, T., Oligny, S.: On the Applicability of COSMIC-FFP for Measuring Software Throughout its Life Cycle. In: 11th European Software Control and Metrics Conference, Munich (2000)
18. Pastor, O., Gómez, J., Insfrán E., Pelechano, V.: The OO-Method Approach for Information Systems Modelling: From Object-Oriented Conceptual Modeling to Automated Programming. Information Systems 26(7), 507--534 (2001)
19. Pastor, O., Molina, J. C.: Model-Driven Architecture in Practice: A Software Production Environment Based on Conceptual Modeling. Springer (2007)
20. Poels, G.: A Functional Size Measurement Method for Event-Based Object-oriented Enterprise Models. In: Int. Conf. on Enterprise Inf. Systems (ICEIS), pp. 667--675 (2002)
21. Poels, G., Dedene, G.: Distance-based software measurement: necessary and sufficient properties for software measures. Inf. and Software Technology 42(1), 35--46 (2000)
22. Selic, B.: The Pragmatics of Model-Driven Development. IEEE Sof. 20(5), 19--25 (2003)
23. Selic, B., Gullekson, G., Ward, P.T.: Real-time Object Oriented Modelling. Wiley (1994)

A proactive process-driven approach in the quest for high quality UML models

Gianna Reggio[1], Egidio Astesiano[1], and Filippo Ricca[2]

[1] DISI, Università di Genova, Italy, astes|gianna.reggio@disi.unige.it
[2] Unità CINI at DISI*, Genova, Italy, filippo.ricca@disi.unige.it

Abstract. Out of our own yearly experience with students' projects and case studies, we propose a pragmatic approach to the production of high quality UML models. That approach is proactive in the sense of being preventive, and is process-driven in the sense that it applies to the various tasks within a development process model. It consists in a meta-approach to be instantiated in every subprocess requiring the production of a UML model. It (i) starts with a method for that subprocess, (ii) uses only a subset of UML with a clear semantics, (iii) adopts a suitable profile and then, to guarantee some basic quality aspects of the models, (iv) defines their metamodel with constraints expressed at varying degree of formality. Moreover, our proposal is put into perspective with reference to the current foundational work on the UML model quality assurance.

1 Introduction

As it has been emphasized by several authors (see, e.g., [1, 2], also for other references), the issue of quality for UML models poses specific problems with respect to the general issue of software quality. First, because they are models, as opposed to source code, they may play different roles in the development process and at different levels of abstraction. Secondly, because they use UML, that is a notation with an extremely rich set of features, often lacking a clear semantics (or even proposing a choice of different semantics) and, finally, very flexible in allowing a lot of freedom in its use.

We have faced those problems in an almost decennial experience of attempts at supporting the production of high quality UML models, by investigating and teaching, experimenting with student projects and, more recently, interacting with people on the industry side (see, e.g., [3]). The lessons learnt span the whole range of the essential quality issues (or dimensions), from syntactic to semantic and pragmatic quality. However, it seems to us that there is a prominent encompassing lesson, namely the paramount relevance of the overall development process, with its associated subprocesses, by which and within which the UML models are built, as opposed to the attention to the production of the single models in isolation. At the end, we will put forward and propose to discuss this lesson.

* Laboratorio Iniziativa Software FINMECCANICA/ELSAG spa - CINI

The need of taking a preventive approach to product quality by considering the overall process has led, for the software development in general, to consider general quality framework aimed at the software development improvement. For the case of UML-models quality the impact of the different development phases has been discussed, e.g., in [1], and a framework for engineering the quality in the overall process has been recently presented in [2].

Our contribution here is more pragmatic (and less ambitious); it stems directly from our experience trying to extract, and formalize to a certain extent, an approach that we have come to use over the years. Our approach is process-driven in the sense that it takes into account the particular subprocess, or task, to be performed in the development process and is proactive in the sense that, for that subprocess, it provides explicit support for the production of high quality models. We have applied our approach to quite different subprocesses/tasks with quite different context constraints:

- requirement specification [4] based on the use case technique, both with textual description of the use case scenarios and with use case fully modeled using the UML. It is interesting to note, how our approach works also for modeling techniques integrating the UML with other notations, in this case natural language artifacts (the scenarios description).
- design specification [4];
- business modeling [5];
- object-oriented software libraries.

In the following section we qualify the essence of our work, providing perspectives and motivations; in the third section we illustrate our approach, first in overview and then in detail; then we outline some recent related works, and finally we offer some conclusions.

2 Motivation and perspective

As in every engineering branch leading to a product, also in software engineering any sensible approach to quality assurance has two facets: evaluation and prevention.

The evaluation of product quality is performed on the basis of some defined quality attributes, usually by means of some metrics. In the software area, since the beginning of the investigations, initially for source code, the quality issue has presented uncommon difficulties, leading to unclear specifications of the attributes, lack of metrics, and sometimes diverging viewpoints. With the emergence of UML as a de facto standard, the software community has first reacted as believing that the use of such powerful and intuitive notation could lead to high quality products. Only quite recently, say in the last five years, it has been realized that the evaluation of UML related quality issues have a complex nature, due on one side to the nature of the artifacts, which are obviously models and not code, and, on the other side, to the peculiarities of the UML. For a comprehensive picture of the issue and an overall quality model for UML

we refer to [1]. Out of that work we single out among other important insights, as particularly relevant to this paper, the emphasis given to the relationship between model properties and development phases.

Product quality evaluation is complementary and preliminary to prevention in the obvious sense that we need to know our quality target to prevent deviations from quality (defects). Prevention may take different forms, but we are particularly concerned with two of them, that we are briefly reviewing. A classical form is the use of behavioral rules to follow in the building of an artifact. In the case of UML model quality a notable instantiation are the "modeling conventions" introduced in [6] "to ensure a uniform manner of modeling and to prevent for defects". The second form we are interested here is a broader view of prevention consisting in an overall attention to guide the software development process. It is well-known that since the mid-eighties various initiatives have gone in the direction of proposing frameworks for improving the quality of the SW products indirectly by improving the development process, the so-called software-process-improvement approaches (e.g., CMMI[3], to quote the best-known). But those frameworks are of coarse grain, so to speak, with respect the specific case of UML-driven development. In the case of UML models a recent paper going in that direction is [2], that proposes a framework for "engineering the quality" in the overall process. We have found in that paper a possible broader theoretical frame for what we have learnt and done, in practice, in some years of experiments with students' projects and a number of case studies with industry people. Indeed we have come to use a pragmatic approach which is proactive in the sense of being preventive, and is process-driven in the sense that it applies to the various tasks within a development process model.

There are three basic assumptions at the root of what we propose.

First, we believe in the importance of precision, in the double sense of defining clearly the kind of model we need and of using UML constructs with a well defined semantics. That assumption is at the basis of what we have called and advocated as well-founded methods (more than "formal methods"), see, e.g. [7].

Second, we always assume that, in performing a specific task within a development phase, a technical method is followed, providing guidelines about how to perform that task. That method could be part of an overall method encompassing the whole development process or a specific method for a phase or a task.

Third, we are well aware, as many have noted, that "precision" has to be intended in relation not only to the phase and task, but also to the used method. As a paradigmatic example, consider the different use of and, consequently, of requirements on the UML models within an agile or an MDA approach. In a similar way, we have to take into account the potential users of the models, who can vary from an expert developer (e.g., in the deployment phase) to a business analyst or a customer. And, of course, it is well-known that within an overall development process, the different phases, say requirements vs. design, suggest different criteria in evaluating the quality of models.

[3] www.sei.cmu.edu/cmmi/

In essence, we have singled out an approach, to be instantiated in every subprocess requiring the production of an UML model, that, starting with a method for that subprocess, only uses a subset of UML with a clear semantics, adopts a suitable profile and then, to guarantee some basic quality aspects (e.g., syntactic, but not only) of the models, defines their metamodel with constraints expressed at varying degree of formality. The result is a modification of that method that includes provisions for quality aspects, both preventive quality and evalutative quality aspects are considered. Our approach also includes a validation of the modified methods, by inspecting the models obtained in the various applications of the approach. However our experience in validation is only related to a our own method MARS (see [3] for a synthetic view and [4] for a complete presentation) and we plan to discuss it in another more experimental paper.

3 Our approach

In this section, we present a two-step approach to drive a developer in the production of high quality UML models. The first subsection is devoted to provide an overview and the second to present the approach in some detail with the help of a running example. In the second subsection we will use as running example the application of our approach to the case of the *object-oriented analysis* (shortly, OOA) proposed by Coad&Yourdon [8] and explained in details in [9]. Recall that our approach starts from a development method, providing guidelines to perform a specific task in the development process.

3.1 Overview

The first (meta) step requires to define the "good models", i.e., of good quality, and to embed in the method the guidelines and the activities helping produce such models. The second (meta)step requires the modelers to follow the now modified method, encompassing quality related aspects.

We present now our approach. Let us assume to have at hand a method METH for performing a task T in a specific development process model, for a specific development problem, whose purpose is to produce a UML model MOD. Our two-step approach is as follows:

MSTEP 1

- Define a meta-model for "good" MOD that we call META. The instances of META will be the UML models of good-quality expected to be produced by the task T.
- Define a quality-enhanced version of METH, that we call Q-METH, taking into account the fact that MOD should be compliant with META; thus Q-METH is a set of steps driving the developer in the production of MOD in accord to META (i.e, a workflow).
- Validate experimentally META and Q-METH.

MSTEP 2

- Follow Q-METH to perform the task T to produce a a good UML model MOD.

The UML activity diagram in Fig. 1 visually presents *MSTEP 1* in a detailed way. The activities (action nodes) are represented by rectangles with rounded corners, while object nodes are depicted by means of rectangles. The object nodes crossing the borders represent the inputs and the outputs of the activity. The various activities are summarized below and then detailed in the subsequent paragraphs of this section.

The input of *MSTEP 1* is a method METH for a specific task producing a UML model called MOD. The first step in *MSTEP 1* is choosing which UML to use for the production of MOD, i.e., to define the UML profile (PROF) to be used for MOD. The second step consists in defining META, a meta-model for MOD with associated formal and informal constraints. This action could raise the need to modify METH, for example, when some parts have to be clarified (see, e.g., the case of the OOA analysis shown later). Then the next step is to modify METH to produce MOD according to META, i.e., making precise the steps to follow for building MOD. The outcome of the first (meta) step is the procedure/workflow Q-METH, which includes a final inspection useful to check that all the constraints defined in META have been respected. In the description of *MSTEP 1* we have included the possibility (a suggestion) of performing an experimental validation of META by inspecting the UML models produced by developers according to META and following Q-METH. As a result, this could lead to a modification of META.

3.2 Exemplified illustration

Let us now illustrate *MSTEP 1* in more detail, also by instantiating METH with the running example OOA method (see Fig. 2), that we call in the following OOAM and whose main ideas are briefly recalled below.

The first task of OOAM is to give a model representing in terms of objects/classes the domain and/or the software to be developed (say SW). Classical techniques, such as the textual analysis, have been proposed to discover the needed objects/classes from textual documents describing the domain of SW and the SW itself. The method suggests starting from a "processing narrative", i.e., a text in natural language defining what SW should do. The processing narrative will be analyzed to find the names, the verbs, and the adjectives. Then, some criteria will help decide which of them will become classes, attributes, operations and associations. The outcome of the task is presented by means of a UML class diagram. Subsequent tasks will lead to define the behavioral aspects of the found classes and to represent them by means of UML sequence diagrams, activity diagrams, and state machine diagrams (in our example we do not consider these activities).

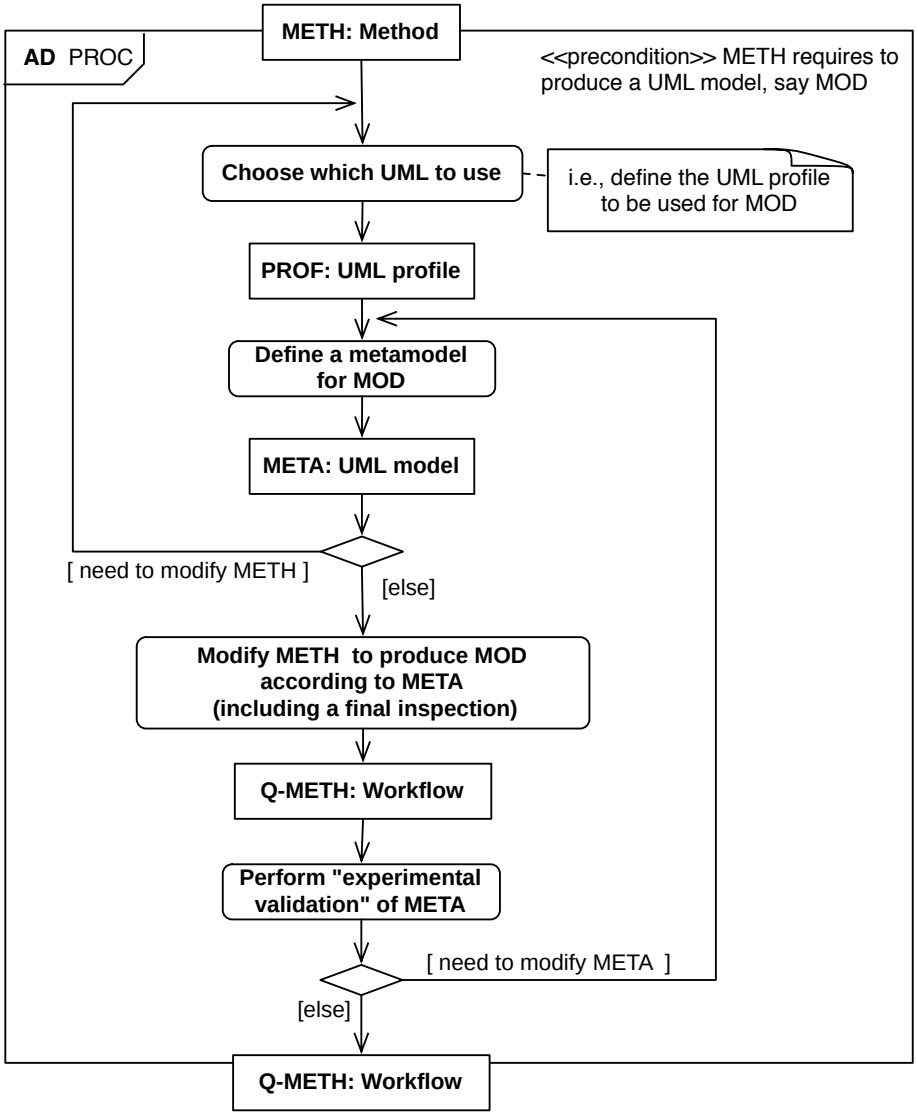


Fig. 1. UML activity diagram describing the activity of *MSTEP 1*

METH	OOAM
T	give an OO model representing the domain (and a SW application) starting from the narrative
MOD	UML class diagrams + constraints
PROF	subset consisting of the main constructs used in the class diagram, plus the convention that the default for the multiplicity of an association is "1"
META	see Fig. 3
Q-METH	see Fig. 4

Fig. 2. Relationships between the ingredients of our approach and the OOAM running example

The method METH. The input of *MSTEP 1* is a method METH showing a specific way to perform a specific task T, and there is a unique precondition: METH requires to produce a UML model called MOD.

Basically, OOAM can be read as: "Given a processing narrative, perform a textual analysis finding names, verbs and adjectives, filter them using some given criteria and determine a set of candidate classes (with attributes and operations), associations and specialization relationships among them. Build a tentative UML class diagram with constraints using such ingredients, and then revise and complete it till to have an object-oriented abstract view of SW expressed using the terminology of the domain".

Define the UML profile PROF. Since no sensible method METH requires neither to produce a generic UML model nor to use all the innumerable features of the UML, we can safely assume that MOD will be defined using a subset of the UML. Furthermore, any sensible method will fix the semantic variation points relative to the chosen subset, and in a very large number of cases, some specific stereotypes will be used in MOD. Thus, we can assume that MOD will be defined using a UML profile, also if in some case that profile will be very light consisting just in defining a subset of the UML and fixing some semantic variation points.

In the OOAM case the used UML profile PROF is simply the subset consisting of the main constructs used in the class diagram, plus the convention that the default for the multiplicity of an association is "1", and thus will not be further specified.

Define META, the meta-model of the good models. The meta-model of the well-formed MOD is defined using UML (or more precisely using MOF[4]) and consists of:

- a class diagram having a (meta) class Mod. MOD is well-formed *if and only if* it corresponds to an instance of Mod;
- some invariant constraints on the classes appearing in that class diagram.

[4] http://www.omg.org/technology/documents/modeling_spec_catalog.htm

The constraints characterizing MOD may be *formal* (expressed using OCL) and *informal* (expressed using the natural language). Constraints can concern the following aspects:

formal

- syntactic well-formedness of the various diagrams parts of the model;
- syntactic consistency among the various diagrams parts of the model;
- completeness (i.e., all the relevant elements of the model have to be present);
- minimality (i.e., a good model cannot have either useless or unused elements);
- limits at the dimension of the diagrams based on some metrics (e.g., the number of attributes in a class is < 15) — the idea is that there are limits to the possibility to "read/understand" a visual representation, thus the quality should also mean to enforce limit on the complexity/dimension of the diagrams (human cognitive abilities permit grasping information related to about 7 objects at most [10]).

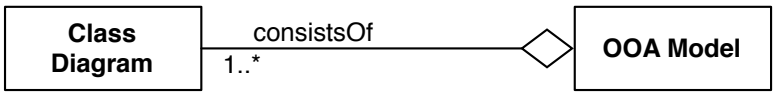
informal

- semantic soundness of the various diagrams parts of the model;
- semantic consistency among the various diagrams parts of the model;
- semantic consistency w.r.t. other models, in the cases that METH takes as inputs other models (expressed using either UML or other notations), produced in precedent tasks; for example consider the case of a task concerning the development of a model of a design, that takes as input a model corresponding to the specification of the requirements;
- naming conventions;
- format of the parts of MOD expressed in natural language;
- layout templates and guidelines (i.e., suggestions, prescriptions, prohibitions about how to give a layout to the various diagrams).

Returning to our example, the meta-model for OOAM is very simple, see Fig. 3. The list of constraints, summarized in Fig. 3, is instead quite interesting.

The last constraint in Fig. 3 poses a problem, since the name of SW will appear in the processing narratives and thus the textual analysis will return a corresponding candidate class with a large number of operations, while a classical recommendation for the OO development is "there is no class corresponding to the system to develop". We can imagine some possible ways to fix this problem. For example, we can require that the class corresponding to SW must appear in the class diagrams, and that it should be related to some other classes by composition (parts of SW) or by plain associations (entities interacting with SW). Furthermore, its operations should be logically organized in various *interfaces*[5], each one covering a specific functionality; those interfaces will lead to the design of the user interfaces in the subsequent steps of the development process. On

[5] A UML interface is a variant of the class construct corresponding to an abstract class (i.e., it cannot be instantiated) and having only operations. It may be then specialized by standard classes realizing its operations.



Type	Constraints
F	An association has either a name or at least one of its association ends is named
F	Attributes, arguments and results of operations must be typed using either predefined types or classes defined in the model (completeness)
F	The only constraint allowed are class invariants, at most one invariant for each class
F	Each class defined in the model should be linked to another class by some association or used to type either an attribute, or an argument or the result of an operation of some other class (minimality)
I	Invariant constraints on classes and multiplicity annotations on associations should be consistent
I	Invariant constraints should be satisfiable
I, P	Naming convention for the classes: an instance of the class CL should be denoted by *cl*. For example, an instance of a class BOOK should be denotable by *book*, and an instance of the class BOOKS should be denotable by *books*, i.e., a class named with a plural name must corresponds to a set/collection/group of entities.
I, P	Attributes should be named using names and adjectives
I, P	The name of an association should such that it is sensible when used to navigate the association in both sense (e.g., father is wrong, familyMember is correct)
I, P	The name of operations should be verbs or verbal phrases
I, P	No class should correspond to SW

(F is for formal constraint, I for informal and P for pragmatic)

Fig. 3. The meta-model META for the OOAM case

the other hand, we can instead require that the software to be developed does not appear in the processing narrative, that thus will be a pure description of the domain in which the software SW will operate, and so the OOAM activity will be a domain analysis.

We choose the second possibility, and slightly modify the method to require that in the input processing narrative there are no references to SW, and that instead it describes the domain of SW in a detailed way.

Define the quality-enhanced method Q-METH. The outcome of *MSTEP 1* is a procedure/workflow called Q-METH that will drive the developer in the production of MOD taking into account META and the associated constraints. In other words, *MSTEP 1* is useful to transform the method METH, usually defined vaguely in natural language, in a well-defined and ordered series of steps (Q-METH) in which the concept of quality is embedded. It is required that the last activity of Q-METH is to perform a systematic inspection of the produced UML model MOD to check that the form expressed by META and the associated constraints have been respected.

In the final inspection it is possible to use software tools to check part of the formal constraints, for example existing (commercial) tools with some capabilities for customization, such as SDMetrics[6].

In Fig. 4 we presents the quality-enhanced method Q-METH for the case of the OOAM.

The job of defining META and Q-METH usually requires to revise and clarify the original input method METH, since the presence of unclear and/or underdefined parts may prevent to define META and Q-METH.

Perform the experimental validation. Once the enhanced method, Q-METH, based on META has been put at work, we should start to collect some data when performing the final inspection guided by META and the associated constraints.

The data to collect are the non-conformance of the produced models with META and its associated constraints and the frequency of their occurrences. In the following, we call a non-conformance a "*violation*". Our years-long experience in using methods defined following *MSTEP 1*, and more in general in teaching UML in a precise way[7], leads us to think that the violations and the rate of occurrence are not evenly distributed. Usually, there are some violations that appear more frequently, some of them with a very high frequency.

Whenever one of these very frequent violations, say V, has been detected, we have to analyze it, since usually there is a very specific motivation for its so frequent appearance. We should examine the models where it occurs and try to see if the violation is related with a "problem" in the model.

[6] http://www.sdmetrics.com/

[7] That is when explicitly the static semantics is defined, also in a more stringent way than that required by the UML specification or enforced by a supporting tool.

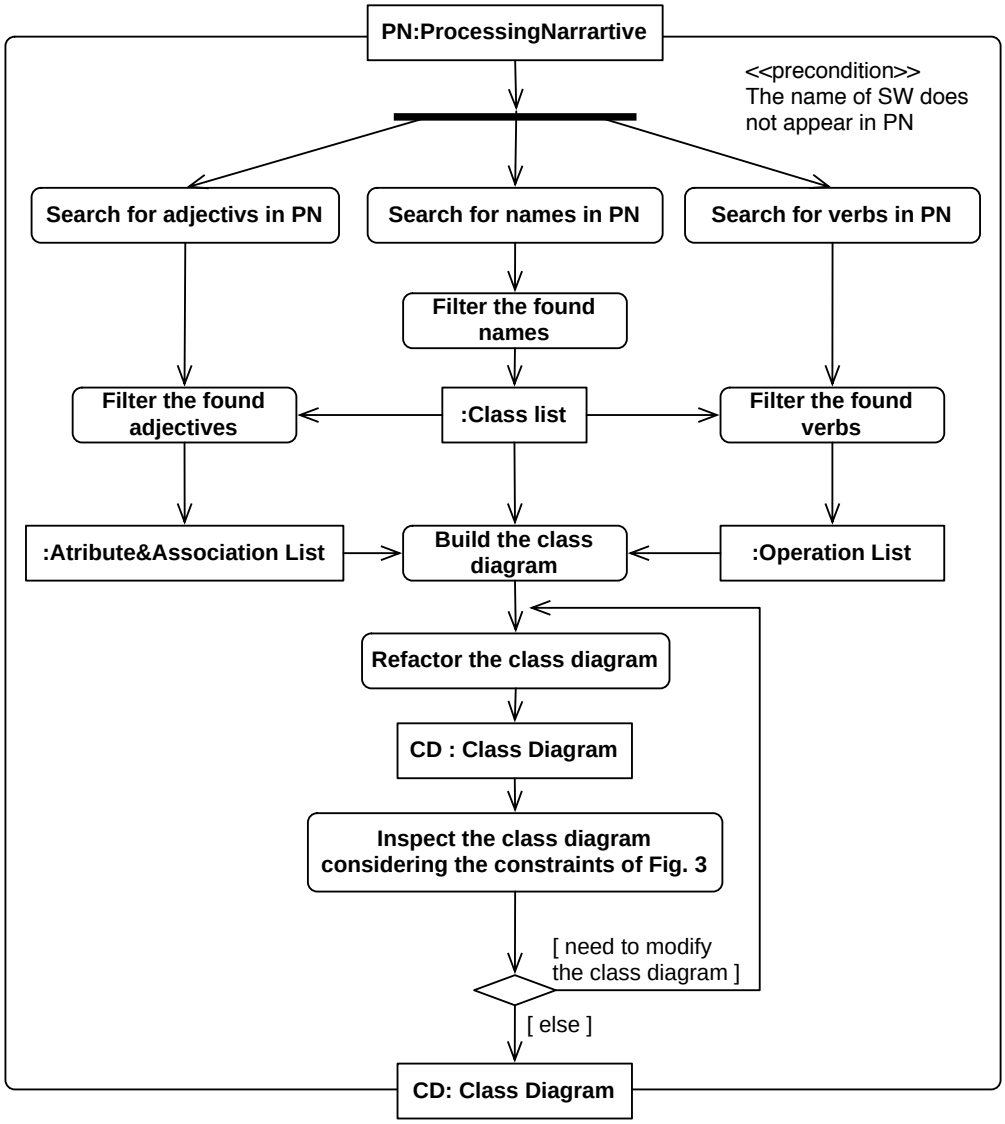


Fig. 4. The enhanced method Q-METH for the OOAM case

The answer may be:

1. the violation is correlated to problems in a significant number of cases
2. the violation is not correlated to problems in a significant number of cases
3. there is a standard and almost automatic way to correct the model to eliminate the violation

In the case of 1, the method Q-METH should be modified trying to focalize the attention of the modeler on this point, and similarly the final inspection part should modified to have the check of violation V in a high position.

In the case of 2, the form of the models META should be modified by allowing to include also the models with violation V, also by modifying the used UML profile PROF. Unfortunately, in some of these cases there is no way to allow the violating models except that by allowing something that it is not a well formed UML model, and so these subcases should be treated as case 1.

In the case of 3, it may be useful to modify either Q-METH to catch the developer's attention to this point, or META to incorporate a better practice for the modelers, or also PROF to offer an improved notational support.

As for what concerns the validation, we didn't have the opportunity to perform it in the case of OOAM. We have instead collected some relevant experience, related to our own method MARS [3, 4], over the years when we have used that method in a yearly course projects. We plan to present the related findings in another more detailed paper.

4 Related work

Several frameworks, approaches, and experiments (surveys, case studies and controlled experiments) have been presented in literature dealing with the problem of defining, improving, monitoring, and evaluating the quality of software models. Some of these works deal with a classification of model quality properties in general, as Lindland does in [11], where he considers the quality of models along three dimensions: syntax, semantics and pragmatics. The first refers to the relationships between the model and the modeling language without considering the meaning of the elements, i.e., a model is syntactically correct if it respects all the constraints forced by the modeling language. The second considers the meaning of the elements of the model, while the last refers at how the audience will interpret the model [11]. The idea is that there are limits to the possibility of "reading/understanding" a model, and thus the quality should also mean to enforce some limits on the complexity/dimension of the models.

The quality of a system is usually assessed using some metrics (e.g., cyclomatic complexity) connected to quality attributes (e.g., complexity of a program) by means of a quality model. Building over and improving some classical work (see their references), in a very interesting paper focused on UML-based Software Development, Lange and Chaudron in [1] present a four-level quality

model, based on several industrial case studies The four considered levels are: (i) the use (e.g., development and maintenance), (ii) the purpose of modeling (e.g., comprehension and communication), (ii) inherent characteristics of the artifacts (e.g., consistency and aesthetics), and (iv) metrics. The peculiarity of the quality model presented in [1] is that it distinguishes between quality characteristics of the system while in the earlier work this distinction is not clear.

A number of other papers address the quality issue of UML models from an experimental viewpoint, either evaluating the impact of best practices, such as the use of modeling conventions, or analyzing the model defects in some real case studies. Modeling conventions are similar to coding conventions, but they apply to the model instead of the code. They are defined by Lange et al. in [6] as "Conventions to ensure a uniform manner of modeling and to prevent for defects". They can belong to several categories: Layout, Naming, Completeness, etc. An example of naming modeling convention is [12]: "Classes, use cases, operations, etc. must have a name and it should be non-ambiguous and precisely express the function/role/characteristic of an element". The effectiveness of modeling convention and their impact is investigated in [6] experimenting with 106 masters' students and by Du Bois et alt. in [12], where a controlled experiment is conducted with 27 master students. The results of that study indicate that modeling conventions decrease the defect density of the model but they are not able to improve clarity, completeness, and validity of the information. The authors interpret this result concluding that mere properties of the model (e.g., syntax, design and layout) are not sufficient to improve its quality. Even if UML is the de facto standard for modeling software systems its lack of a formal semantics and its complexity cause the risk of a lot of practical defects. Lange and Chaudron in [13] conduct a study trying to quantify the distribution of defect types in real industrial models. The study shows that the number of defects found in industrial UML models is very large. The most common are [13]: multiple definitions of classes or use cases under the same name, large numbers of classes and interfaces without operations, messages in sequence diagrams that do not correspond to operations in class diagrams, or messages without names. The authors conclude that prevention techniques such as modeling conventions, training and the use of tools could contribute to improve the situation.

All papers quote above are concerned especially with model quality in the sense of the product quality, namely classification and analysis of the properties of the final product, sometimes in relation to the best practices used in the development. Quite a different view is taken by Mohagheghi and Dehlen in [2], where the notion of Model-Driven Quality Engineering (MDQE) is put forward and advocated, emphasizing in particular the need of "engineering the quality" together with "evaluating the quality" in the various phases of the development. We feel much sympathetic with the attention those authors have to the process and believe that we could insert our approach within their framework, though we are not strict in following a purely transformational model-driven approach.

5 Conclusions

Like many other early users and promoters of the use of the UML, we have passed through the typical phases of the adoption of a new notation, indeed a special one, because so rich and complex and in evolution. Inevitably, the quality effort in the beginning was focused on the syntactic correctness, the clarification of the semantics and the selection of the semantically clear constructs. The next phase was devoted to provide what, in the light of [6], we can call modeling conventions, and also some suggestions on pragmatic issues. Then, in the most recent years, also stimulated by the emergence of some theoretical work in the field of UML model quality assurance, we have reflected on our own quality approach trying to systematize it. We have come out with a meta approach that, starting form a development method for a task in a phase of the development process related to a specific problem, defines the models to be produced by a metamodel with constraints. The approach we have come to is based on the assumption that "precision", in the sense of a clear definition of the kind of models we need for a specific task following the various constraints, will result in better models. Note that for us not always precision means "very detailed"; depending on the method, the context, the problem to solve and the task, even models with loose parts may be admitted. In this respect, it is interesting to mention that the basic development method that we have proposed and used, namely MARS [4], offers, almost for every task, two alternatives modalities, light and precise. The light alternative, that admits parts in a natural language, has been actually adopted in a business requirements case study performed in a collaboration with a company.

Admittedly, we have not yet addressed the issue of the rigorous "quality model" to be used as reference in a task, rather relying on the "quality by experience" model that many developers still use in their work. In our view this aspect should be addressed and included in our approach along the line, e.g., of the quality engineering framework of [2] and the nice quality evaluation model of [1]. This last model could provide a guide also in the validation process of the modified method. Currently our validation experience with MARS for the main phases and tasks has not the status of a scientific assessment; we have only a qualitative confirmation based on the course projects of the last three years.

6 Acknowledgments

Filippo Ricca was supported by the project "Iniziativa Software" Finmeccanica (Research unity of Software Engineering coordinated by Prof. Egidio Astesiano and situated c/o DISI, Genova) funded by Elsag-Datamat.

References

1. Lange, C.F., Chaudron, M.R.: Managing model quality in UML-based software development. In: 13th IEEE International Workshop on Software Technology and Engineering Practice (STEP'05), IEEE Computer Society (September 2005) 7–16

2. Mohagheghi, P., Dehlen, V.: An overview of quality frameworks in model-driven engineering and observations on transformation quality. In: Workshop on Quality in Modeling at MODELS 2007. (September 2007)
3. Astesiano, E., Cerioli, M., Reggio, G., Ricca, F.: A phased highly-interactive approach to teaching UML-based software development. In: Educators' Symposium at MODELS 2007. (September 2007)
4. Astesiano, E., Reggio, G.: MARS: Model-based Adaptively Rigorous Software development. Technical Report DISI–TR–2007–12, DISI – Università di Genova, Italy (2007) Available at `ftp://ftp.disi.unige.it/person/ReggioG/MARS01.pdf`.
5. Astesiano, E., Ricca, F., Reggio, G.: Modeling Business within a UML-based Rigorous Software Development Approach. In Degano, P., Nicola, R.D., Meseguer, J., eds.: Concurrency, Graphs and Models Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday. Number 5065 in Lecture Notes in Computer Science. Springer Verlag, Berlin (2008)
6. Lange, C., Bois, B.D., Chaudron, M.R.V., Demeyer, S.: An experimental investigation of UML modeling conventions. In Nierstrasz, O., Whittle, J., Harel, D., Reggio, G., eds.: MoDELS. Volume 4199 of Lecture Notes in Computer Science., Springer (2006) 27–41
7. Astesiano, E., Reggio, G.: Towards a well-founded UML-based development method. In: Conference on Software Engineering and Formal Methods, SEFM 2003. (22-27 September 2003)
8. Coad, P., Yourdon, E.: Object Oriented Analysis, 2nd Edition. Yourdon Press Computing Series (1990)
9. Pressman, R.S.: Software Engineering: A Practitioner's Approach, 6nd Edition. McGraw-Hill (2005)
10. Miller, G.A.: The magical number seven, plus or minus two: Some limits on our capacity for processing information. The Psychological Review **63**(March) (1956) 81–97
11. Lindland, O.I., Sindre, G., Solvberg, A.: Understanding quality in conceptual modeling. IEEE Software **11**(2) (1994) 42–49
12. Bois, B.D., Lange, C.F., Demeyer, S., Chaudron, M.R.V.: A qualitative investigation of UML modelling conventions. In: Workshop on Quality in Modeling at MODELS 2006. (1-6 October 2006)
13. Lange, C., Chaudron, M.R.: Defects in industrial UML models — a multiple case study. In: Workshop on Quality in Modeling at MODELS 2007. (September 2007)

Description and Implementation of a Style Guide for UML

Mohammed Hindawi, Lionel Morel, Régis Aubry, Jean-Louis Sourrouille

Université de Lyon
INSA Lyon, LIESP, Bât. B. Pascal, 69621 Villeurbanne, France
{Mohammed.Hindawi, Lionel.Morel, Régis.Aubry, Jean-Louis.Sourrouille}@insa-lyon.fr

Abstract. Model quality is still an open issue, and a first step towards quality could be a style guide. A style guide is a set of rules aiming to help the developer improving models in many directions such as good practices, methodology, consistency, modeling or architectural style, conventions conformance etc. First, this paper attempts to clarify the meaning of notions being used such as rule or modeling domain semantics. Then, several examples illustrate a possible classification of rules, and the verification process is detailed. A style guide is not universal: each project manager should be able to customize his/her set of rules according to specific needs. In addition to rules expressed in OCL, we describe a user interface to facilitate the specification of rules based on quantifiers, along with the translation of these rules into OCL.

1. Introduction

In the emerging context of Model Driven Engineering, software development more and more focuses on models. On the other hand, the software engineering community has known for a long time the advantages of early fault detection. Thus to check models at the beginning of the development cycle appears a promising direction. For a reader, models have a meaning related to the application domain, but generally, a model checker only knows the semantics of the modeling domain. As a result, application domain semantics is a matter for users, while tools may help for modeling issues. Beyond faults, which are all the more difficult to find that models are imprecise and abstract, many model properties are of interest for developers.

A style guide is a set of rules aiming to help the developer improving models in many directions such as good practices, methodology, consistency, modeling or architectural style, conventions conformance etc. Some rules are hints while others are warnings, which means that they are potential errors. These rules check “good properties”, which are kinds of quality criteria. However, the quality of a model is relative to application requirements, and errors are ignored as long as they do not go beyond the quality objectives that have been set according to requirements. Conversely, a style guide checker notifies all the rule violations: the developer defines his/her own objectives and priorities, often based on error gravity.

Developers could be in charge of rule checking. However, in practice, only automated checks are suitable not to increase developer burden, but also because

manual checks are unsure. Consequently, as many rules as possible should be given a formal description, and only rules expressed in natural language will require manual checks. There is no universal style guide. Each development team may have its own needs depending on applications, hence we need an easy way to specify rules. UML provides OCL as a description language, but non-experts find it difficult to use. This implies the need for specific tooling to describe and manage a set of rules, and to control the verification process that should be as flexible and automated as possible.

The rest of the paper is organized as follows: section 2 gives definitions and attempts to clarify the meaning of used notions; section 3 classifies some rule descriptions; section 4 shows the verification process, defines the main tool components, and details the user interface to specify rules including the translation into OCL. Then we discuss related works and conclude.

2. Context and definitions

This section describes the context of the work and defines some notions used in the rest of the paper. Moreover, we aim to clarify what it means to apply rules to a model.

2.1. Syntactic vs. Semantic Correctness

In software engineering, a model is a representation, from a given point of view, of a system expressed in some formalism [4]. The formalism definition includes notions and their semantics. This semantics induces constraints on the model, for instance the semantics of inheritance induces that cycles are not allowed along the inheritance relationship. A model expressed in a formalism is *correct* when it conforms to all the constraints of this formalism. The UML specifies constraints in both OCL and natural language. We call the former *syntactic* constraints and the latter *semantic* constraints[1] [16][6]. A model that meets syntactic constraints is *syntactically correct*, and a model that meets semantic constraints is *semantically correct*. Syntactic constraints can be checked automatically while semantic constraints are left to human users, hence it is not possible to check automatically whether a model is correct or not. In everyday cases, UML models are at best syntactically correct but their semantic correctness is unknown. In addition, semantic variation points in the UML specification require human choices. For instance, the choice of the communication policy of a UML *Port* leads to different valid communication sequences: “If several connectors are attached on one side of a port, then any request arriving at the other side of this port will be forwarded on all links *or* only one link...” [17].

In the following, we consider only syntactically correct models. Additional constraints aim to increase the semantic correctness.

[1] Although surprising, this definition has the advantage of being precise: even in programming language, the difference between syntax and semantics is unclear. Going deeper into this issue does not help due to the lack of unambiguous difference between semantic and syntactic constraints expressed both in OCL. Anyway, the reader may think of semantic constraints as constraints specified in natural language.

2.2. Interpretations

A system can be modeled in different ways. A model *interpretation* is defined as the meaning of this model in a semantic domain. A model has generally several interpretations in a semantic domain (Fig. 1), but the set of interpretations of an incorrect model is empty in any domain. The natural semantic domain of a modeling language such as UML is the *modeling domain*. There is no consensus about the semantics of a universal modeling domain; hence, we assume that there are several modeling domains, each one with its own semantics, e.g., active objects do not behave the same according to modeling contexts. As all modeling languages, the UML does not meet all modeling needs, while on the other hand it allows expressions that the semantics of modeling domains may forbid, e.g., to send a signal to a set of objects that cannot catch it. Further, the relationship between the semantics of the modeling domains and the semantics of UML is an important issue, but it is out of the scope of this work. The modeling domain is not to mix up with the application domain. In the modeling domain, a class *Dog* may inherit from a class *Bird*, but in the application domain, this inheritance relationship is surely wrong.

An interpretation is *licit* in a semantic domain when it has a meaning in that domain, i.e., it conforms to the semantics of this domain. A UML model can be both correct and illicit. For instance, a *TypedElement* without *Type* is correct in UML, but when the element is the receiver of a message, it is illicit in most modeling domains.

Refinement. The number of interpretations of a model evolves during the development. An abstract model has a large number of interpretations due to the lack of details. Along the development process, models are refined and become more and more complete and precise; hence, the number of interpretations decreases (Fig. 2a). For example, an undirected association between two classes *A* and *B* has three potential interpretations: *AB* is navigable, *BA* is navigable or *AB* and *BA* are both navigable. Navigability restriction at a further modeling step may reduce these

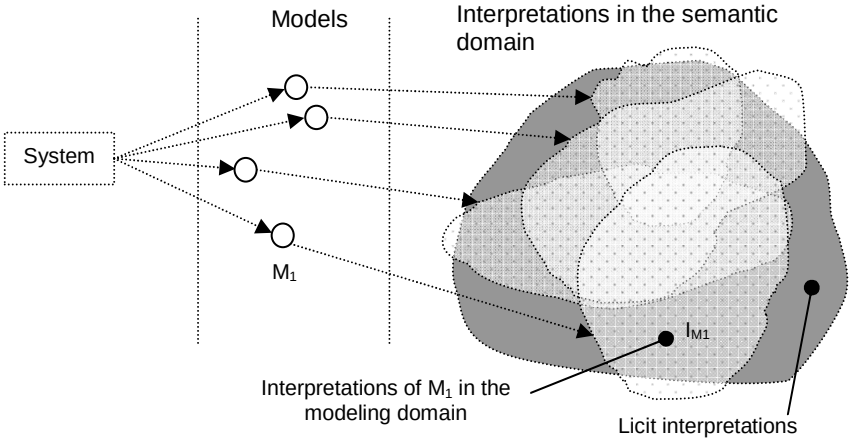


Fig. 1. "System – Models" and "Model – Interpretations" relationships

interpretations to only one. At the end of the refinement process, one interpretation is selected to generate code. This code is a model whose interpretations form an equivalence class from the developer point of view, i.e., all the interpretations are equivalent: the expression $a+b+c$ can be interpreted as either $(a+b)+c$ or $a+(b+c)$.

Checking models. Within a domain, a model is *consistent* when it conforms to the semantics of this domain. Hence, a correct UML model that is consistent in a modeling domain has at least one licit interpretation. There is no formal definition of the semantics of modeling domains, but many works propose consistency rules to check that models meet some semantic domain constraints (650 rules in [14]).

Modeling domain constraints that can be expressed in a formal language are easy to check. The outstanding issue is how to check models to meet semantic constraints? There is no fundamental difference between semantic constraints from UML and from the modeling domain. Both are expressed in natural language, both apply to models, and modeling domain constraints aim to reject models with no licit interpretation. A first idea is to define *consistency rules* that are stronger than the actual semantic constraints, but that we can express in a formal way. These rules reduce the expressive power of UML (Fig. 2b), i.e., reject potentially correct models and forbid some interpretations, but it is the price for automating checks. A second idea is to define rules that will help the developer to make well-formed models. These rules forbid model expressions leading *generally* to models with illicit or questionable interpretations. Finally, human reviews help finding problems that formal rules cannot detect. At code generation, model analysis will reveal errors but it is too late. The checking process fails when an error that was visible in a model is discovered at run time.

2.3. Style guide

A style guide defines a set of rules that any model must conform to. Style guides reduce the number of acceptable models and force developers to make models owning the wished properties, which results in smaller sets of licit interpretations (Fig. 2b).

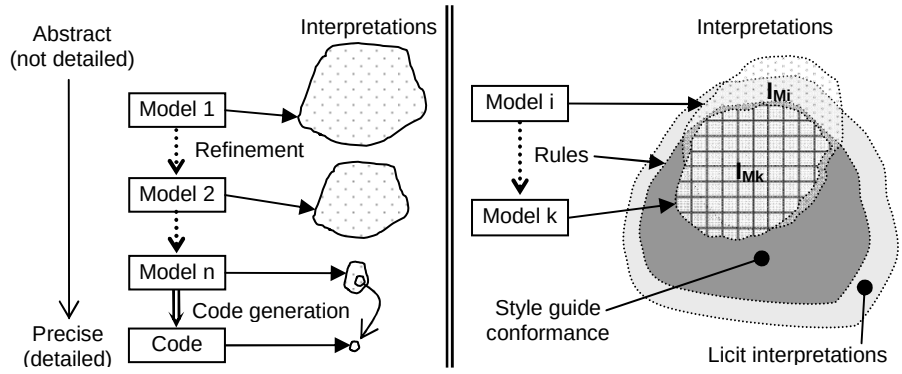


Fig. 2. (a) Model evolution and interpretations (b) Reduction of interpretations

Unlike language constraints that should not be violated, bypassing rules that are simple hints is allowed. In addition to consistency rules, we consider the following kinds of rules:

- Language conventions are rules agreed within a group,
- Guidelines aim to help developers making well-formed models,
- Methodologies induce constraints aiming to make better models from given points of view. They force activities order, model form, deliverables, etc. These rules depend on the chosen methodology,
- Good practice rules come from experience in specific domains, including modeling domain, for instance developers know for a long time that low coupling between elements is better.

Checking model conformance to a style guide is usually a human issue. Due to the high cost of human reviews, the return on investment seems unsure. To reduce the human burden, this paper details an approach to describe and implement an automatically checkable style guide. We define an architecture on top of existing tools, and a checking process. The description of a style guide goes beyond a paper because there are too many rules. Moreover, some are common to most users while others depend on needs that are context specific.

2.4. Style guide and Quality

The style guide defines the boundaries of the set of models owning the wished properties. Beyond model correctness and consistency, the style guide aims to help developers designing models with a better quality, for instance using good practices. From this viewpoint, a style guide is a key element in the quality management process. Of course, the quality of a model that does not meet all the style guide rules is not definitely low, hence the link between rule violation and quality is to be clarified.

Within a multilevel framework for quality assessment such as ISO9126 [8] or [11], rule violation is at the level of metrics. Metrics are aggregated to form attributes, which in turn are aggregated to form characteristics. The quality of a model is not subject to conformance to some individual rules, but rather to some statistical knowledge embodied as threshold values for attributes and characteristics. These thresholds come from quality objectives that are set according to specific needs of applications. From the quality point of view, only deviations from these values will lead to corrections, otherwise the model has the expected quality. While the style guide notifies all rule violations, non-quality is detected only when the combination of a set of metrics reach critical thresholds.

Both style guide and quality assessment detect failures, i.e., non-quality. The software engineering community usually applies the hidden rule “a model with no failure is good”, but nobody knows to what extent a model is good. To ease model comparison, each rule has a gravity level: some violations are just warnings or hints while others are serious errors. Developers know that no serious rule violation should remain while some warnings are acceptable.

Research about software quality in usual programming has reached a high degree of knowledge and skills for a long time. Software managers definitely know the impact of software quality. In spite of this high theoretical maturity level, code quality remains an under-exploited way to improve software. The main lessons learned from surveys show that quality should be provided at no cost, with a suitable support, and should not induce delays in the project. Thus we should pay a great attention to the implementation of the style guide: integration of the verifier within the modeling tool, very simple checking process, flexible user interface, easy rule description, etc.

3. Rules

3.1. Identifying and classifying rules.

Models are checked along several *dimensions* corresponding to different software engineering areas such as methodology, good practices, or modeling. The semantics of each area induces rules. Since generally this semantics is not described, experts from these areas are in charge of rule identification. The dimension is our main structuring property of rules. In addition, each rule owns a set of properties aiming to explain it, to give further comments, to specify gravity, to link it with model parts or development process stages, to specify and implement it, to describe correction actions, etc. These properties are needed at any moment, for instance to classify violations according to their importance, to organize and manage rule description, to help the developer dealing with errors. We give below examples of rules classified by dimension, although rules might often be attached to several dimensions. As mentioned above, some syntactic rules can be stronger than the actual semantic constraints to allow writing them in OCL. Rule descriptions are deliberately short and sometimes imprecise due to available space:

- **Methodology** rules come from method description, e.g., “The application domain model is mandatory”, or “Any communication between actors and subsystem goes through an interface class” (USDP [9]). This latter rule aims to limit changes to a set of well-identified classes when communication protocols between actors and subsystem are modified. The UML itself induces methodology rules, e.g., “Each use case describes at least one scenario to be specified as a sequence diagram”.
  Within the development process, methodologies distinguish steps or *phases* [15] such as requirement elicitation, elaboration, or detailed design. Whatever the methodology, these phases are required to identify moments in the life cycle of artifacts, and as a result to identify levels of abstraction. Each part of a model can be in a different phase. The phase is used to select the set of rules to be applied to each part of a model at a given moment. Beyond phases, to make a distinction between Platform Independent Models and Platform Specific Models allows detecting model expressions that are forbidden at a given stage of the development, and helps keeping the wished independency level.
- **Common methodology** gathers rules that applies whatever the methodology. They come from skills of experienced developers, e.g.: “A black box sequence diagram only holds actors and a subsystem (definition)”, “A white box sequence diagram

holds objects, ports and components and the only actor that triggers the initial stimulus (definition)" or even "A black box sequence diagram only holds communications between actors and a subsystem, not between actors".

- **Consistency** rules detect meaningless expressions in the modeling domain, e.g., "The initial stimulus in a sequence diagram is triggered by an *Actor* or a *Port*, i.e., neither a class instance nor a *Component*", or a usual one "Navigability: any message in a sequence diagram is sent to an accessible receiver, either method parameter or attribute/role in an association". Based on redundancies in the model, some rules detect inconsistencies, e.g., "Within a sequence diagram, actor-to-subsystem interactions should correspond to associations between actors and use cases of this subsystem".
- **Modeling style** rules detect expressions that are *generally* meaningless in the modeling domain. Unlike consistency rules, breaking these rules is tolerated, e.g., "Within any complete class model, a path through navigable associations should link the root class to any class (not a database schema)", or "A sequence diagram is triggered by only one stimulus which is the first in the chronological order". The former rule requires marking the *root* class in the model. The latter rule reduces (apparently) the expressive power but ensures some good properties for the model (no simultaneous waits). Similarly, the rule "Each *ConnectableElement* (from metamodel) in the sequence diagram should be either a port or a class instance" reduces the expressive power forcing components to be connected through ports.
- **Completeness** rules check missing elements from mandatory or even usual links between model elements, e.g., "When the subsystem *B* is an output actor of the subsystem *A*, then *A* should be an input actor in the description of the subsystem *B*", or "Each association actor-to-use case should be implemented in at least one sequence diagram describing a scenario of this use case".
- **Good practices** rules are often hints, e.g., "Cycles along class associations are to be avoided" which aims to reduce coupling, or "To specify systematically bi-directional navigability for associations in a final class model (just before code generation) is likely unnecessary".
- **Conventions** rules are group agreements about syntactic forms, e.g., "Any public name should be capitalized". Within contexts such as education, to meet convention rules is often mandatory, e.g., "When the class of an attribute is represented on the same diagram, drawing the association is mandatory" to avoid hidden associations. Unlike in [12], conventions are limited to a narrow field since we have many other dimensions.
- **Architecture style** rules aim to aid developers to meet software architecture styles such as low-coupling/high-cohesion or Model-View-Controller, e.g.,"A view knows its model but the model does not know its views". Unusual architectures can be detected, e.g., "A subsystem should not appear on its list of actors".
- **Refinement** and **trace** related rules aim to check consistency along the development cycle and to enforce links between model elements at different stages, e.g., "A sequence diagram should be associated with a use case or a less detailed sequence diagram (traceability)".
- **Specification gap** rules deal with non-standard UML. As mentioned before, we consider only models that conform to the UML syntactic specifications (Fig. 3). Modeling tools often allow expressions that do not conform to the UML

specifications. To deal with this issue, a set of specific rules fulfills the gap between each tool and standard UML specifications. This dimension avoids mixing up style guide additional constraints with UML syntactic constraints that tools do not check. To specify these rules is clearly the work of a UML expert.

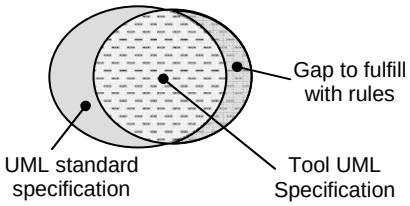


Fig. 3. Standard vs. tool UML specification

Since rules check a large variety of issues, errors resulting from their violation do not have the same gravity. We classify the violations into three categories: *error*, *warning* and *hint*. When a consistency rule is violated, the model has no meaning in the domain of modeling and the violation is an error to be corrected. A violation that might result in a further problem is a warning, and the correction is likely to be preferred. When rules such as methodology are hints for a better modeling process, their violation reduces the quality but to correct them after model completion is not always desirable. Although there is a strong link between dimensions and gravity categories, the gravity is not attached to the dimension: experts and/or project managers set the suitable value.

Finally, to avoid experts specifying rules again and again, a set of standard/common built-in rules should be provided by tools implementing the style guide. Thus, only specification gap rules and customized rules are to be specified.

3.2. Expressing rules

First, rules are expressed in natural language. Next, they have to be formulated in a formal language, preferably OCL. OCL has a power of expression equivalent to first-order logic, but as a main drawback, non-experts generally find it difficult to use. To allow non-experts to formalize rules, we propose a graphical approach on top of OCL

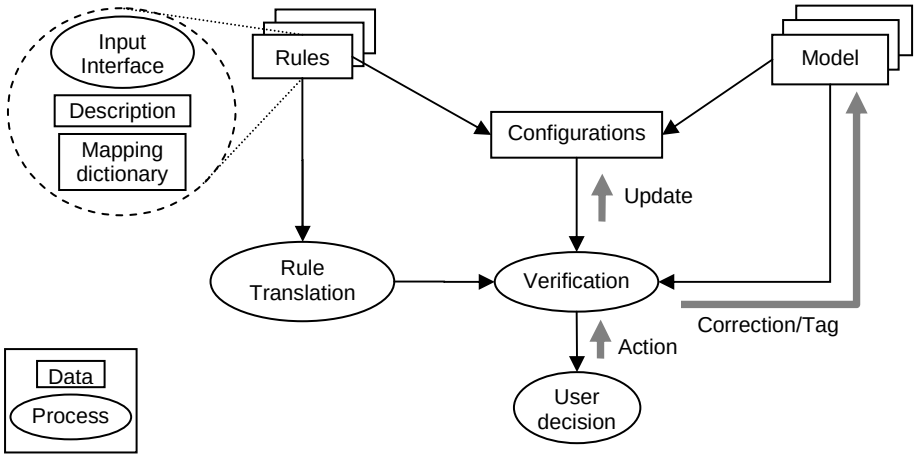


Fig. 4. Verification process.

that rely on generally well-known notions of first-order logic quantification. The rule "Each connected element in a white box sequence diagram should be either a port or an instance of a class" is written:

$\forall x \in \text{WBSeqDiag}, \text{IsConnectableElem}(x) \Rightarrow \text{IsPort}(x) \vee \exists y, \text{IsClass}(y) \wedge x.\text{class}=y$

where *WBSeqDiag* is a collection of model elements, *IsT*(*x*) is a predicate which is true when *x* is an instance of *T*. The general form of this rule is:

$\forall x \in X, R_1(x) \Rightarrow R_2(x) \vee \exists y, R_3(y)$

Based on quantifiers, the interface provides a limited set of standard forms that ease description but whose expressive power is lower that the OCL one.

The main remaining issue is the link between model notions such as object, class or interaction, and UML metamodel notions. To read and understand the meta-model is hard and reserved to UML experts. In the implementation section, we propose an approach based on rewriting rules that makes it easier to use metamodel notions.

4. Verification Process

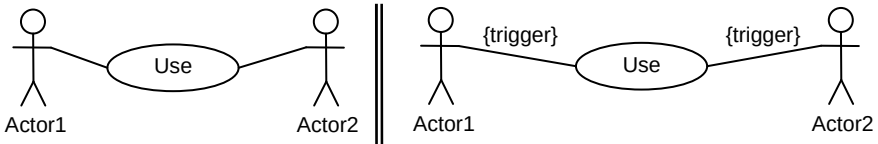
The verification process lies on the architecture illustrated in Fig. 4. The set of rules to check depends on the role of the user, the phase in the development process, the kind of model, temporary choices of the developer, etc. Links between rules, model and users are expressed through *configurations* that control the verification process.

To check the style guide and to ensure traceability, we need to annotate the model with data such as the phase or the root class. On the other hand, the implementation of the verification process requires marking models. During the development process, developers regularly check models and occasionally, they are not interested in some types of errors because they focus on other aspects. Thus, marks on model elements specify which rules should be checked. To summarize, we need two types of model tags: adornments to complete the model description, and error-processing tags to control the verification process. UML *taggedValues* are suitable for both purposes.

4.1. Architecture and Process

The Fig. 4 gives the main processes and data of the verification process:

- **Rules** are managed through a user interface described below. Rule properties are stored in a description file separating common rules and properties from specific ones. The mapping dictionary maps rule names expressed in natural language to metamodel notions or OCL expressions (detailed further table 1).



a) Warning: No triggering actor **b) Error**: Two triggering actors

Fig. 5. Two violations of the same rule

- A **configuration** links together a model and a set of rules that we call a rule package. Packages are defined either using rule properties, e.g., phase *Elaboration*, dimension *Good practices*, gravity *error*, or manually for specific needs. For instance, a team member may decide to keep watch on a particular set of rules. Within rule packages, flags signal rules not to check temporarily.
- The **verification** is directed by configurations and results in messages and actions. When the checking result is *failed*, data about the related rule and model elements are displayed. According to the rule, several choices are offered: to annotate the model with a tag, to invalidate the rule either in the configuration or in the model, to automatically correct the model. Let us take an example to show the verification process for the rule "All initial stimuli (triggers) of a use case come from the same actor". When no trigger is specified (Fig. 5a), checking results in a warning, assuming that the developer does not still complete the model. When two triggers are specified (Fig. 5b), checking results in an error because two actors trigger the use case.

In this particular case, the same rule has two *diagnosis* according to the number of triggers: 0 is a warning, and greater than 1 an error. More generally, a rule can check several properties of an element. Diagnoses avoid several descriptions of the same rule, but each diagnosis holds its own message, gravity, correction, etc.

4.2. Implementation

To check easily the rules and to aid correction, the style guide is embodied in an IDE tool that supplies all the required services for a quicker implementation (the implementation is an ongoing work, we choose to implement on top of Eclipse). The integration into one tool reduces the cost of training and use, and simplifies the checking process. We need plug-in extensions to check rules, to manage errors, to manage configurations, and to manage corrections using model transformations, but the tricky point is rule description. The style guide implementation should provide a set of common built-in rules that users may select through customized configurations. The proposed user interface allows specifying the rules. This section focuses on this interface aiming to hide the trickiest aspects of OCL.

To provide a simplified description of constraints in OCL while keeping the same expressive power is difficult. Our approach is a compromise: the simpler constraints are specified through the provided interface, while the remaining ones are to write directly in OCL. The main form of the user interface (Fig. 6) provides fields to define rules, which allows to expressing a subset of all the possible OCL expressions. We illustrate the description interface with the rule: "Each connected element in a white box sequence diagram should be either a port or an instance of a class". An equivalent expression using quasi-natural language could be: "For any element *e* in a white box sequence diagram, for any connected element *e*, either *e* is a *Port* or *e* is an instance of a class". The later expression is close to first-order logic and its structure fits well with our generic input form that reads as follow:

For any *Sequence diagram* in *Model diagrams* **such as** *White box* is true
    For any connected element *e*
        e is a *Port* **or** *e* is an instance of a *Class*

The next step is the translation into OCL. Quantifiers like interface operators are translated into OCL operations such as *forall*, *select*, *exists*, etc. Notions such as *Sequence diagram* or *Class* are to translate into notions of the UML metamodel. Splitting the OCL rule into small expressions will ease reading.

Translation of "For any *Sequence diagram* in *Model diagrams* **such as** *White box* is true"

First, UML does not supply the notion of diagram: sequence diagrams are *Interaction* owned by packages. From *Package*, the interactions are (Fig. 7):

```
self.ownedMember->select( i | i.oclIsKindOf(Interaction) )
```

We extract model packages from the metaclass *NamedElement*:

```
NamedElement::allPackages(): Set(Package) ; -- standard operation
    allPackages = NamedElement.allInstances->select( p | p.oclIsKindOf(Package) )
```

Selecting *Interactions*:

```
NamedElement:: sdFilter() : Set(Interaction) ;
    sdFilter = allPackages()->iterate(p ; result :Set(Interaction)={} |
                result->union(p.ownedMember->select( i | i.oclIsKindOf(Interaction) ) ) )
```

The UML does not provide the user defined notion of *White Box* sequence diagram, which means that the developer has to specify the kind of *Interaction* with a UML TaggedValue kindOf = {BlackBox, WhiteBox, Final}. The operation

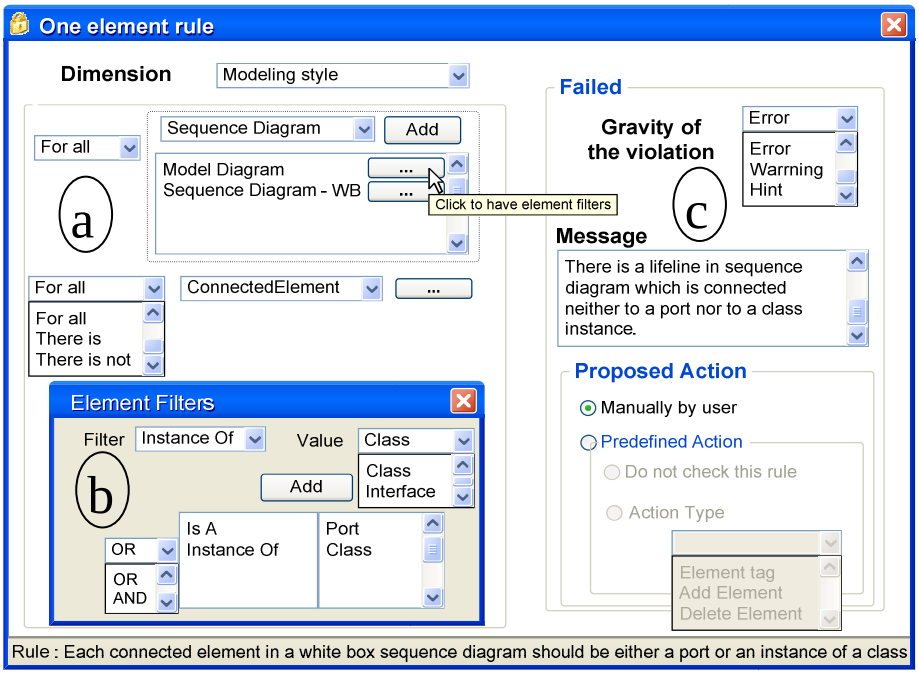


Fig. 6. Rule description interface

GetKindOf('WhiteBox') returns *true* for a WhiteBox *Interaction*. The set of *White Box* sequence diagrams is:

```
NamedElement:: sdWBFilter() : Set(Interaction) ;
    sdWBFilter = sdFilter()->select( i | i.GetKindOf('WhiteBox') )
```

Translation of "For any connected element *e*, *e* is a *Port* **or** *e* is an instance of a *Class"*

From *Interaction*, the form of the path to access to a *ConnectableElement* is:

```
    lifeline[f].represents      -- another rule checks whether connectable element exists
```

From *ConnectableElement*, either the element is a *Port* or the *Property* is typed with a *Class*:

```
    oclIsKindOf(Port)  or type.oclIsKindOf(Class)
```

From *Interaction*, the complete expression is:

```
    lifeline->forAll( f |
            f.represents.oclIsKindOf(Port)  or f.represents.type.oclIsKindOf(Class)   ) )
```

OCL final constraint:

```
NamedElement:: Rule() : Boolean ;
    Rule = sdWBFilter()->forAll( i | i.lifeline->forAll( f |
            f.represents.oclIsKindOf(Port)  or f.represents.type.oclIsKindOf(Class)   ) )
```

The interface is aided to avoid any syntactic error. The user selects values in lists, e.g., when *Model Diagrams* is selected (Fig. 6a), the next filter list supplies only allowed subsets. When *Sequence diagram* is selected, the filter only allows *BlackBox*, *WhiteBox* and *Final* (leaf according to the trace relationship). The translation of the interface expression into OCL is based on a rewriting principle (Table 1): each element in the list has a value in the mapping dictionary. We plan to build the dictionary from an aided interface that lists all the accessible item names in the

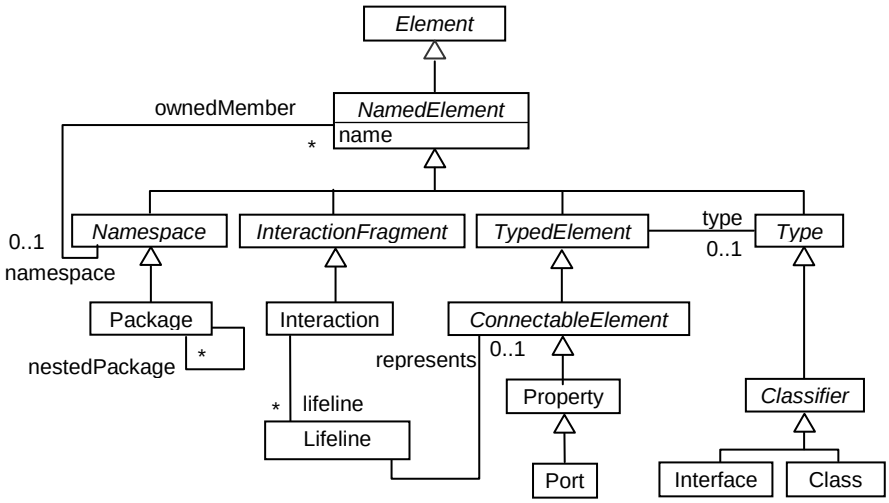


Fig. 7. Root of the required metamodel (from [17])

context. For instance in the metamodel Fig. 7, from *Interaction* the three only choices are *name*, *lifeline* and *namespace*. This work is close to the definition of a subset of UML [].

The right area of the interface (Fig. 6c) deals with additional properties and corrective actions when the checked element does not conform to the rule. During the verification process, the user is prompted to choose an action that may eliminate or temporarily hide errors. At the end of the process, the remaining violations are displayed: not to disturb the user, checks are only on demand. A command allows removing the model tags and configuration flags that prevent error messages.

4. Related works

Rules. Style guide rules come from various sources. The UML specification [17] is the first information source. Some UML books such as [1][2] include recommendations or style guides that help making "better" models. Methodology books such as RUP [15] or USDP [9] also provide tips and rules. Modeling conventions in [12] correspond to several dimensions within our classification. These modeling conventions proved to be useful to reduce model "defects", which confirms that a style guide is an important issue. In addition, papers related to rules or metrics for UML models are interesting sources [13]. Obviously, a complete style guide description requires a large space [14].

Implementation. A tool may enforce built-in rules that cannot be changed, which relieve from the burden of rule description but prevent customization. A template defines a framework, for instance related to a methodology (e.g., RUP [15]). The user cannot go out of the frame, but remaining dimensions are not checked. "The experience shows that templates are helpful, but they do not ensure that the model as a whole is complete" [7]. To summarize, templates enforce a subset of the required rules only, therefore a preferable way will be to include this subset into a more

Table 1. Mapping dictionary

Name	OCL expression
Model diagrams	NamedElement:: Rule() : Boolean ; R1 = allPackages() - - built-in operation, diagrams are owned by packages
Sequence diagram	R2 = R1->iterate(p ; result :Set(Interaction)={} | result->union(p.ownedMember->select(i | i.oclIsKindOf(Interaction))))
White Box	R3 = R2->select(i | i.GetKindOf('WhiteBox'))
Connected element	Rule = R3.lifeline->forAll(f ; x:ConnectableElement= f.represents | R5)
Is A	R5a = x.oclIsKindOf(Port)
Instance Of	R5b = x.type.oclIsKindOf(Class)
or	R5 = R5a **or** R5b

flexible solution.

When rules are written in natural language, the verification of the style guide must be done manually. The description of rules within books such as [1] comes into this category. Works aiming at automating the verification process should express rules in a formal language. The automated verification on demand is the best solution but proposals are still rare [5][7]. In [7], a checker prototype fully automatically verifies models from rules described using a specific language. Although rule description is different, this work is close to our project. We agree with [5] and many others that find it difficult to write rules in OCL. Instead of defining a new language as in [7], we provide a user interface to aid specifying rules that are next translated into OCL. This way we keep a standard language while aiding rule description. In this direction, some works aim to facilitate OCL writing: VisualOCL [3][10] visualizes OCL expressions in an alternative notation. It provides additional information, which increases the usability of OCL. However, to use such tool implies experience in OCL. We try to overcome this issue by proposing an interface easy to use, at a high abstraction level, but rather far from OCL, which implies an additional and tricky translation process.

5. Conclusion

This project is under development[2] and some issues are still pending. The advance of our solution lies in the integration of several technical artifacts to form a complete methodology and tooling. This integration associated with automated checking and style guide customization is a necessary condition for actual use in companies. Some particularly relevant elements in our approach include:

- Selective checking of model parts using tags, which avoid re-checking of rules and messages related to incomplete model parts, therefore lighten the user burden;
- Selective checking according to the current phase in the methodology;
- Customization of the set of active rules in a configuration file according to developer role and experience, application domain, expected “quality”, etc.
- Aid for correcting models: when a rule is violated, the developer may choose a predefined action including model change by applying patterns;
- Aid for defining rules: the graphical interface helps project managers in the definition of rules for their own style guide.

This work is part of a grant aiming to assess model quality. The companies involved in the project will help us to tune quality assessment from metrics. Model quality assessment is relative to application quality requirements and developers do not always know the important quality criteria. A style guide brings the educational aspect needed to help increasing models’ “good properties”: it detects all rules violations but also provides hints, warns to avoid potential errors, and may include company know-how. Finally, a style guide is a quite necessary complement to put into practice quality assessment.

[2] Partly financed by the grant PACTE QUALITE with the Rhône-Alpes regional government.

References

1 Ambler, Scott W., “The Elements of UML 2.0 Style”, Cambridge University Press, 2005
2 G. Booch, J. Rumbaugh, I. Jacobson: “The Unified Modeling Language User Guide” Addison-Wesley, 1998
3 P. Bottoni, M. Koch, F. Parisi-Presicce and G. Taentzer: “A Visualization of OCL using Collaborations”. UML 2001, LNCS 2185, Springer, pp. 257–271.
4. G. Caplat, J.L. Sourrouille, "MDA: Model Mapping using Formalism Extension", *IEEE Software*, Vol. 22(2), pp.44-51, 2005
5 Farkas, T.; Hein, C.; Ritter, T. : ”Automatic Evaluation of Modeling Rules and Design Guidelines”, proc. of the Workshop “From code centric to Model centric Soft. Eng.”, http://www.esi.es/modelware/c2m/papers.php
6. D. Harel, B. Rumpe, "Modeling Languages: Syntax, Semantics and All That Stuff", TR MCS00-16, The Weizmann Institute of Science, 2000.
7 Hnatkowska, B., “Verification of Good Design Style of UML Models”, Proc. Int. Conf. Information System Implementation and Modeling, 2007, http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-252/paper10.pdf
8 ISO, International Organization for Standardization, "ISO 9126-1:2001, Software engineering – Product quality, Part 1: Quality model", 2001
9 Jacobson I., Booch G., and Rumbaugh J., *The Unified Software Development Process*, Addison-Wesley, 1999.
10 C. Kiesner, G. Taentzer, and J. Winkelmann. Visual OCL: A Visual Notation of the Object Constraint Language. Technical Report 2002/23, Tech. Univ. of Berlin, 2002
11 L. Kuzniarz, L. Pareto, JL. Sourrouille, M. Staron, "The 3rd Workshop on Quality in Modeling", Models in software engineering, LNCS 5002, Springer, 2008, pp.271-274
12 C.F.J. Lange, B. DuBois, M.R.V. Chaudron, S. Demeyer: “Experimentally investigating the effectiveness and effort of modeling conventions for the UML”, CS-Report 06-14, Tech. Univ. Eindhoven, 2006.
13 Malgouyres, H., Motet, G., “A UML model consistency verification approach based on meta-modelling formalization”. SAC 2006: 1804-1809
14 H. Malgouyres, J.P. Seuma-Vidal, G. Motet: “UML 2.0 Consistency Rules”, V 1.1 (in french) http://www.lesia.insa-toulouse.fr/~motet/UML/CoherenceUML_v1_1_100605.pdf
15 Rational Unified Process, IMB Corp. 1987 (2008).
16 Sourrouille, J.-L., Caplat, G.,"A Pragmatic View about Consistency Checking of UML Model", Work. Consistency Problems in UML-Based Software Dev., 2003, pp.43-50.
17 UML, "OMG Unified Modeling Language", Version 2.1.2, 2007
18 JL Sourrouille, M. Hindawi, L. Morel, R. Aubry, "Specifying consistent subsets of UML", Educators Symposium @ MODELS'08, 2008 (extended version http://www.if.insa-lyon.fr/liesp/~sou/Reports/sUML-RR2008_1.pdf)

A Combined Global-Analytical Quality Framework for Data Models

Jonathan Lemaitre and Jean-Luc Hainaut

Computer Science Faculty, FUNDP, Namur, Belgium
{jle,jlh}@info.fundp.ac.be

Abstract. In the domain of data model quality two independent approaches can be identified. The first one proposes a global view mainly based on quality models and frameworks, focusing on high level quality characteristics such as minimality, maintainability and evolvability and on metrics for measuring them. A second research track has concentrated for decades on the analysis of specific problems, ranging from unnormalized structures to unsatisfiability. The latter proposes means for formalizing, detecting and correcting particular defect patterns. Both of these approaches address data model quality issues, but in independent ways. In this paper, we present an attempt to address database schema quality through both approaches in a common framework. We summarize the main concepts and reasoning basis of a project devoted to database schema quality. We propose an operational framework that combines the contribution of both global and analytical views of quality. Our global view focuses on defects categories to evaluate schema quality and error side effect. Our analytical view translates into detection and correction methods of these defects. The final purpose of this work is to propose a precise, intuitive and easy to use quality management methodology for database schema.

1 Introduction

Quality has become one of the major topics in software engineering. Research and industrial communities acknowledge that such concepts as maintainability, portability or evolvability translate in technical terms users satisfaction and economical stakes. The question has been at the core of software engineering for more than three decades. In the nineties, authors have already assessed the impact of poor quality and errors made during the modeling phase [1]. During the last few years quality of models became more and more important owing to the increasingly popular MDE approaches mainly relying on modeling and models transformations.

Looking at data model quality, one can make the distinction between two research approaches. The first one comprises proposals allowing a global assertion of the schema quality through such key concepts as quality models, frameworks and metrics. Quality models are mainly composed of definitions of global quality characteristics [2, 3] while frameworks define particular views and/or contexts

of use for the characteristics [4–6]. These authors define metrics that provide a numerical evaluation of the characteristics of a particular schema. On the other hand some authors study very specific problems as, for example, normalization [7–9], visualization [10, 11], satisfiability [12] and more generally the intra-model consistency [13, 14], etc. They propose a limited but formally defined view of model quality that includes precise problem identification techniques and problem solving.

As far as model quality is concerned, both research approaches have their advantages and limitations. The first one provides an abstract, fast but imprecise global evaluation of a model that can translate into, e.g., development and maintenance time and cost. The second one leads to a precise identification of intuitive classes of structural problems and to their solution, but is of no help when a global evaluation is required. Though they both address the problem of data quality problem, they have been so far developed independently of each other.

The goal of our research is to build a framework that relies on both global and analytical approaches. The result that we would like to obtain is a framework easily applicable and intuitive. This framework will propose methods and tools for addressing specific structural problems and assessing schema quality. Quality assessment will also be associated with particular correction methods. This framework will be valid for data models of future software systems as well as for models of existing, and even legacy, software systems.

Our research focuses on persistent data structures, that is, on database schemas. Though this proposal is independent of the schema abstraction level (i.e., it covers both PIM and PSM levels), we will base the discussion on conceptual examples expressed in a variant of the ER formalism [15].

This paper is structured as follows. First, we recall some aspects of global (section 2) and analytical (section 3) approaches. Section 4 presents our proposal for unifying the these approaches. In section 5 we illustrate our approach with an elementary study of a specific quality characteristic, namely understandability. The last section (6) includes first conclusions and future work.

2 Global approaches of schema quality

In this section, we address informally the main characteristics of the global approaches to schema quality. Quality frameworks and models can be classified into identifiable categories:

– Quality models: they may also be viewed as hierarchical frameworks since they propose a tree-structured view of the quality. The root concept is the global quality which is divide into different global aspects of the quality. Each of these global aspects can also be refined into more specific aspects. The leafs are generally associated with metrics. One of the most famous example is the ISO9126 standard [2]. In this standard, the first level is the quality. The second level copes with such characteristics as maintainability or efficiency. The third level details sub-characteristics depending on one

characteristics, e.g., stability and analyzability contributing to maintainability. In the domain of the database schema, similar hierarchical models have been proposed, e.g., by Hoxmeier [16]. Quality models may also be associated with other kinds of frameworks. For example, some quality models express the links between the characteristics [17, 4] or the relationships between the characteristics and the actors of the modeling process [4]. Quality models have the advantage to propose a structured view, but as quality terms are often at a high abstraction level, they may prove difficult to apply. Furthermore definitions often are ambiguous due to the lack of agreement on the meaning of essential terms.

- Formalization framework: frameworks of this category are quite uncommon. They propose a methodology for building quality metrics using mathematical properties the metrics have to satisfy [18, 19]. Thus these frameworks may not provide quality definition, but a way to enhance the validity of new metrics.
- Causality framework: rather than proposing a hierarchical view of quality, some authors highlight the influence of some quality characteristics on others [17, 5, 20]. This kind of frameworks seems to have more practical use than the others but often require costly empirical validation that provides precise numerical coefficients that measure the characteristics correlation. Moreover, these models address a limited number of quality characteristics. In this category, we can mention the work of Kesh [17] and Maes and Poels [5].
- Semiotic framework: these proposals started with the work of Lindland and al. [21]. They are designed for conceptual model in general and they are rooted in the study of sign processes and detail quality into syntactic, semantic and pragmatic aspects. Later, Krogstie made a new proposal [6] integrating additional aspects such as physical quality and social quality. These frameworks give a *constructivist world-view* [6], i.e., they represent the situation of quality aspects with the related actors of the modeling process (e.g., model, language, user). Other proposals were made occasionally but are very close to the framework of Krogstie [4, 22]. These frameworks have the advantages to represent the modeling context and to link the quality with it. Unfortunately they also stay at a very theoretical level without proposing easily usable means for assessing the quality.

As main advantages of quality frameworks, we can underline the global assessment of the quality and the structuring of the reasoning induced by the frameworks. Nevertheless quality characteristics have different meaning across different frameworks. The frameworks also stay at a theoretical level that impairs their understandability and/or usability.

Considering the data model domain, several metrics are available. Among the different proposals the complexity of the metrics expression may vary from simplistic to overly complex. Metrics are based on counting particular schema objects such as entity types, attributes, relationship types, is-a hierarchies, etc. Using empirical studies, the authors assess the value of some quality characteristics. These values are associated with the result of the simple object counts,

which, for example, may be used to define linear or quadratic polynomial functions using objects count results as parameters. As example we may underline the work of Piattini et al. [23–25] concerning UML class diagrams metrics based on structural properties. Metrics may also be included in a more global view of quality represented by a quality model [26]. The simplest metrics functions are shaped as $\Sigma_{i=1}^{n} a_i x_i$, where a_i is a coefficient and x_i a simple objects count. The expression below presents an example of a more complex metrics, where $ASvsC$ is the Associations versus Classes metrics, N^{AS} is the number of associations in an UML class diagram and N^{C} is the number of classes [27]. Typically, the $ASvsC$ score has to lie between, e.g., 0.3 and 0.6. A result below 0.3 indicates a lack of relationships between classes while a result above 0.6 probably testifies to a lack of modularity (spaghetti-like schema).

$$\boxed{ASvsC = \left(\frac{N^{AS}}{N^{AS} + N^{C}}\right)^{2}}$$

A metric may evaluate quality characteristics like the clarity or expressiveness [28] or low level properties like structural properties [27, 29]. A global overview of metrics is, as for the frameworks types, out of the scope of such a paper, but the following advantages and limitations may be expressed. Metrics are directly applicable and easily usable. However they often are unintuitive and are very costly to validate. This induces difficulties for interpreting results. Furthermore, metrics quality evaluations are based on occurrence frequencies of specific objects in a schema and the comparison with the authorized value intervals of these frequencies. Thus metrics give only a global quality result that hardly allows users to locate some precise defects they can correct in a schema.

3 Analytic approaches to schema quality

An analytic approach concentrates on specific types of defects. Such defects can be formally detected. Their harmfulness has been studied and correction techniques have been proposed. Many proposals have been made for the detection and the correction of specific defects located in models. Some of them may be general enough to concern most of the software product types, such as syntactic errors. Others concern only specific models types as for example the normalization which address data schema and was originally designed for the database logical schemata [7–9]. Defects can be classified according different aspects, e.g., syntax, semantics, readability or maintainability. In our work, we focus on the structural defects, that can be formally identified by schema analysis. The problems related to the application domain semantics have not been considered. We have also discarded visualization aspects [10, 11] (distance between objects, shape, color, etc.) and concrete syntax of the models (e.g., complying with the graphical convention of a specific editor) which can be dealt with independently. However, as data models imply a graphical representation, the visual aspect

cannot be ignored. Its influence and the way it is taken into account will be mentioned in Sec. 5.

Two types of defects can be highlighted. The first includes the *normalization defects* which are *structures in the schema that suggests, and sometimes even scream opportunities for transformations, considering specific requirements like the readability, the evolvability, the performance, etc.*. This definition is derived from the bad-smells definition given by Mens and Demeyer [30]. The second type of defects are the *correctness defects* which comprises *the errors that prevent the schema to be translated into a physical schema or to meet users expectations.* Uninstantiable structures form an important class of errors: they are schema constructs for which there is no valid instances [13, 12]. As we focus on defects patterns, we may underline the recent work of Wahler [31]. He proposes a pattern approach for defining and detecting UML-OCL constraints inconsistencies. Normalization defects (ND) are awkward or inappropriate structures that don't make the schema incorrect or not instantiable. A change in the schema is not mandatory in opposition to the correctness defects (CD) that have to be corrected. Examples in figure 1 give an example of two CD and one ND. Schema (a) violates a syntactical rules stating that the super-type and one of its sub-type may not have attributes with the same name. Schema (b) contains a semantic error due to unsatisfiable cardinality constraints (the only finite population satisfying `COURSE` is empty). Schema (c) is correct but includes an is-a relation with only one sub-type but a partition constraint (symbol `P`). Normally, the super-type and sub-type should be merged since they have the same population.

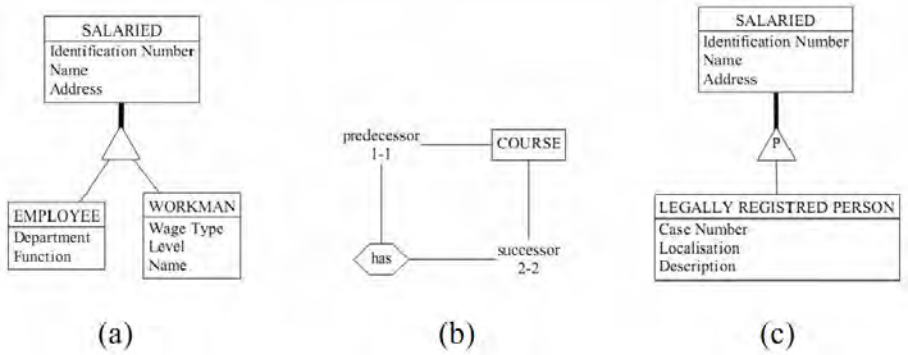


Fig. 1. (a) ER schema containing syntax error. (b) ER schema containing a semantic error. (c) ER schema contains an abnormal construct.

Let us finally mention the class of *unexpressive* or *insufficiently expressive* structures. They are correct structures whose semantics could be better expressed through more appropriate structures. Two examples are very common, namely *attribute entity types* (simple properties expressed as entity types instead of attributes) and *relationship entity types* (associations expressed through en-

tity types instead of relationship types). Unexpressive structures may be considered as ND that may reduce the conciseness of the schema, hence its readability, and finally its maintainability. Eick [32] propose a first approach on data schema understandability and its enhancement using transformations. Rauh and Stickel [33] also proposed a transformation based solution for normalizing ER schemas. Finally, Assenova and Johannesson also studied the schema readability and propose a solution for restructuring a schema based on transformations [34].

As compared with global approaches, which produces metrics based on frequencies and ratios of simple objects (entity types, attributes, relationship types, is-a links, ans so on), analytical approaches study object patterns that generally are complex and semantically rich, but without attempting to count them. In addition, these patterns are most often associated with correction transformations that can be used to improve the quality of the schema.

4 Proposal of a combined approach

The goal we have chosen to reach in this research is to design a quality evaluation and improvement framework for database schemas. In particular, by integrating the contributions of the global and analytical approaches, we expect (1) to augment global approaches with metrics based on semantically rich structural patterns considered as defects, (2) to associate with each structural pattern correction transformations that can be either suggested or automatically applied. By targeting precise structural defects, we expect more informative metrics which better describe the quality of the schema.

Practically, we have structured our work in three steps. The first step is the identification of defect families, formalized by generic patterns resulting from various domains of database theory such as relational normalization, conceptual normalization, satisfiability, redundancy techniques, empirical (good) practices,etc. The second step consists in integrating these patterns into global quality frameworks, hence improving existing metrics systems. The third step addresses quality improvement. This process mainly relies on transformational techniques [35].

The framework we are building is made up of four components.

- A defect taxonomy. Each of the families mentioned in Section 3 are decomposed into more specific categories. The *correctness defects* family comprises two categories, namely syntactical errors and inconsistent constructs. The *normalization defects* family includes seven categories: non minimal constructs, unexpressive constructs, abnormal constructs, irregular constructs, redundant constraints, redundant structures, internal redundancies, presentation defects and standard violation. Each defect of each category is precisely defined by a structural predicate with which the schema can be parsed to identify defect instances. Such predicates can be expressed in some sort of constraint language such as UML OCL [36] or through logic-based languages as described in [35]. In addition, each defect receives a practical documentation comprising an informal description, the conditions, the paradigm and

the abstraction level of the schema where it generally appears as well as some representative examples.
- A limited set of quality characteristics. These characteristics are drawn from standard global approaches proposed in the literature. They are linked to the defects families and categories. The links indicates the influence without using precise ratio factors in order to keep the framework as simple as possible. The rationale of this influence is also explained.
- Assessment methods for the quality characteristics. For assessing the global value of a characteristic we propose a simple counting method based on weights. The weights are declared in the properties of the specific defects description. Assessment also includes a relative evaluation for equivalent structures. Considering a quality characteristic and two different but semantically equivalent structures for expressing the same concepts, the structure with the higher weight for the characteristic would be a better choice, all other weights being equal. A validation procedure is being experimented with the help of a limited team of database design experts. The experts are asked to sort semantically equivalent structures according to their preferences. This procedure seems to bring important advantages compared with usual global approach validation processes: the expert have to compare and to evaluate small structural patterns and not complete schemas, their evaluation is reusable since they are domain independent and finally, the requested effort is quite small (typically half a day).
- Correction methods for the defects detected. When it is possible, corrections methods and changes advises are proposed for improving schema quality. If the correction is obvious and there is only one possible choice, the change can be applied automatically. In the other cases, a list of solutions is propose to the users of the framework. Defects correction will rely on transformational techniques. Model transformations is one of the main baseline of the MDE approach and is known since years in the database domain [35]. This approach follows that of Assenova and Johannesson [34] though we also use non semantics-preserving operators.

5 First application on schema "understandability"

As a first illustration, we will discuss the concept of understandability. There is so far no agreement on a common definition even though the main idea is accepted by most authors. Table 1 collects some of the most popular definitions.

Those definitions are very abstract, so that the authors cannot provide detail about how to evaluate the understandability of a software artifact in general. Except for the last definition that refers to design and structure, the understandability may be considered under various contexts: adequate choice of name, appropriate visual organization, design complexity, etc. Understandability metrics may compute the total time required by the reader for understanding the schema. Unfortunately, this is an "a posteriori" metric. An "a priori" global evaluation is a lot more difficult to develop since human ability and experience differ among users.

Table 1. Understandability definitions

Definition	Author(s)
A software requirement specification (SRS) is understandable if all classes of SRS readers can easily comprehend the meaning of all requirements with a minimum of explanation. Readers include customers, users, project managers, software developers, and testers.	Davis et al. [3]
The understandability is the capacity of the software product to be understood, learned, used and attractive to the user, when used under specific conditions.	ISO/IEC 9126 [2]
An SRS is understandable if all classes of SRS readers can easily comprehend the meaning of all requirements with a minimum of explanation.	Krogstie [6] inspired from Davis et al. [3]
Understandability is defined as the ease with which the concepts and structures in the data model can be understood.	Moody [4]
The properties of the design that enable it to be easily learned and comprehended. This directly relates to the complexity of the design structure.	Bansiya and Davis [26]

In the framework we are developing, we consider the understandability of a schema, disregarding visual aspect, as follows: **We use the global definition of understandability given in the ISO/IEC 9126 norm [2]. We consider that the schema has to be understood by persons who are familiar with the modeling language and its best practices. Hence we consider as a factor of understandability, the adequacy of the schema constructs used with respect to such good practices.** In other words, when the language offers different constructs for expressing a definite concept or fact type, we suggest to use the construct with the highest weight of adequacy according to the reference expert team. Considering the analytical approach, we identified different categories of structures transmitting the same information but using different structural means for that. For each category, experts may associate an understandability score to the patterns. We illustrate this process using the Is-A Partition category.

Figure 2 gives four semantically equivalent but structurally different schemas. Those schemas express the following facts:

1. `A` has a `A1` and a `A2`;
2. `B` owns a `B1` and a `B2`;
3. `C` has a `C1` and a `C2`;
4. `A` is either a `B` or a `C`.

In schema (a), the fact 4 is expressed by an is-a hierarchy in which super-type `A` has two subtypes `B` and `C`. The is-a relationship is defined as a partition, represented with symbol `P`. In schema (b), the fact 4 is represented using relationship

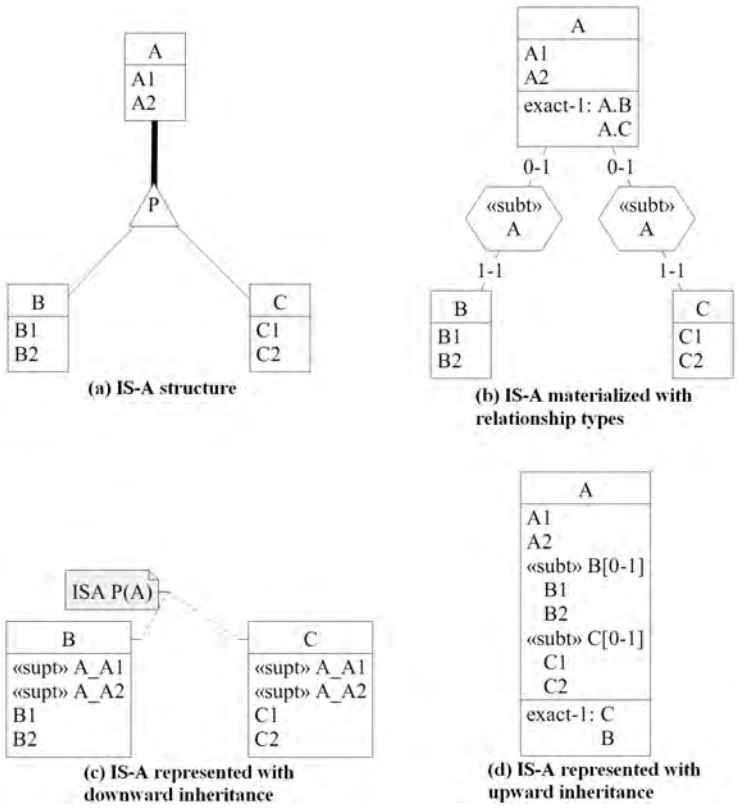


Fig. 2. 4 semantically equivalent structures of the is-a category.

types with the stereotype [1] *subt*, representing the is-a, and an "exactly-1" constraint standing for the partition. Schema (c) represents the information with the downward inheritance, meaning that `A` is materialized in `B` and `C`. The stereotype *supt* highlights the attributes of the supertype. The textual note indicates the type of the is-a. Finally, the schema (d) stands for the upward inheritance which materialized the subtypes into the supertype. The subtypes elements are marked with the stereotype *subt*. As in (b), the "exactly-1" constraint represents the partition.

Obviously, schema (a) complies with the best practices in conceptual modeling, while schema (b), (c) and (d) come closer to lower abstraction level models, e.g., the relational model. Those representations of the concept of category/subcategory have been evaluated respectively *very good*, *average*, *bad* and *average*

[1] Stereotypes are surrounded in the schemas by "<<" and ">>" signs

for (a), (b), (c) and (d) by our research team [2]. The used scale is composed of 5 values : *very bad*, *bad*, *average*, *good* and *very good*. This scale is also considered as an ordinal scale. Indeed, the difference of quality between two values is hardly quantifiable. Plus, the values are discreet and comparisons between values are authorized(*very bad* < *bad*, etc.).

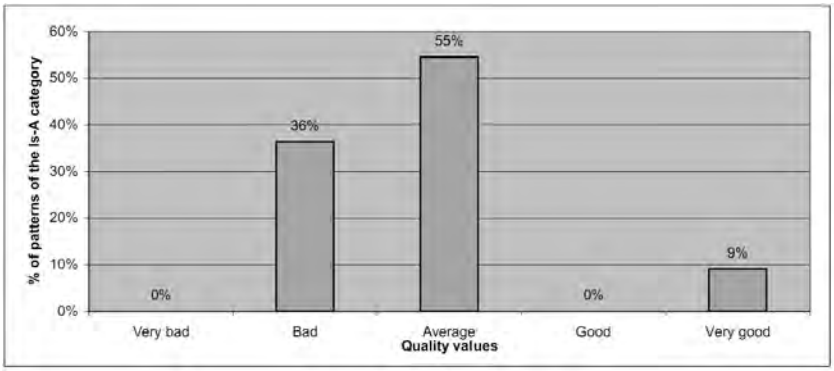


Fig. 3. Graphical representation of the results of the 1st metric applied on a fictive example.

Scores are used to compute the global understandability of the schema for the structures categories. The first metric will give the proportion of *very bad*, *bad*,...in the schema for a category. This metric respect the properties of the ordinal scale used for scoring. The metric highlights understandability problems, but with a composed result. Fig. 3 propose a graphical representation of possible results.

A second metric may be proposed by attributing numerical values to the scores, e.g. *very bad* = 0, *bad* = 1, etc. The weights are transformed to 4, 2, 0 and 2, respectively for (a), (b), (c) and (d) in Fig. 2. This allows us to compute a global understandability measure of any schema for a structures category: for each construct of the schema of the same category, we note the actual weight and the maximum weight. By summing the former on the one hand and the latter on the other hand, then by dividing the first sum by the second one, we get a global understandability measure in the range [0-1]. If the value of this average is close to 0, it indicates a poor quality, while a result close to 1 indicates a good quality. Obtained results are aggregated and easy to read. However, they are violating the ordinal scale properties.

Interestingly, there exist semantics-preserving transformations that produce schema of the Fig. 2 from each other. These transformations are triggered by the

[2] Our research team is composed of 4 people. Without taking into account the years of study, experience in the database field of the team is: one has 30 years of experience, another has 10 years and the 2 last have 5 years.

detection of an instance of a source pattern. By selecting the equivalent pattern with the highest weight, we can automatically fix bad smell defects.

6 Conclusion and future work

This paper introduces a quality framework based on specific data models defects. It derives from the merging of two independent approaches, namely global approaches based on metrics, and analytical approaches that study defect categories and their corrections.

By this framework, we improve the precision and the acceptability of global metrics. In addition, we make it possible, not only to evaluate quality characteristics of a schema, but also to improve them.

As shown in [35], transformations are completely specified by pre- and post conditions, so that they can be implemented in CASE tools. We have developed an extension of the DB-MAIN CASE tool which identifies defect patterns in a schema, and which suggests possible improvement.

This work started in early 2007, so that several problems and questions still have to be studied. We mention three of them, on which we are currently working:

- **What are the interactions between the different quality criteria?** As the causality frameworks express it, the quality criteria influence each other. This influence has to be made explicit. For avoiding this problem, we will try to obtain a limited set of criteria with disjoint view of quality.
- **May the improvement of a structure change the quality of adjacent structures?** Transformations of a structure may influence adjacent structures. It means that improving a part of the schema may decrease the quality of another part. This has to be made clear and detailed in the transformation properties.
- **Is the automated process a better choice?** One of the main goal of the quality discipline is to obtain automatable processes. However this is not realistic when design expertise is required. As we focus on formalizable problems, the patterns and transformations may be implemented into an modeling tool but some transformations choices will have to be selected manually. As an example, the correction of errors or the choice between transformation having equivalent quality results need human intervention.

References

1. Standish Group: Chaos: A recipe for success. Technical report, Standish Group International (1999)
2. ISO/IEC: ISO 9126-1:2001, Software engineering - Product quality, Part 1: Quality model. ISO/IEC (2001)
3. Davis, A., Overmyer, S., Jordan, K., Caruso, J., Dandashi, F., Dinh, A., Kincaid, G., Ledeboer, G., Reynolds, P., Sitaram, P., Ta, A., Theofanos, M.: Identifying and measuring quality in a software requirements specification. In: Proceedings of the First International Software Metrics Symposium. (1993) 141–152

4. Moody, D.L., Shanks, G.G.: Improving the quality of data model: Empirical validation of a quality management framework. International Journal of Information Systems **28** (2003)
5. Maes, A., Poels, G.: Evaluating quality of conceptual models based on user perceptions. In: International Conference on Conceptual Modeling - ER 2006, Tucson, Arizona (November 2006) 54–67
6. Krogstie, J.: Integrating the understanding of quality in requirements specification and conceptual modeling. SIGSOFT Softw. Eng. Notes **23**(1) (1998) 86–91
7. Codd, E.F.: Normalized data structure: A brief tutorial. In: SIGFIDET Workshop, ACM (1971) 1–17
8. Codd, E.F.: Further normalization of the data base relational model. IBM Research Report, San Jose, California **RJ909** (1971)
9. Kent, W.: A simple guide to five normal forms in relational database theory. Commun. ACM **26**(2) (1983) 120–125
10. Moody, D.: Dealing with "map shock": A systematic approach for managing complexity in requirements modelling. REFSQ (2006)
11. Moody, D.: What makes a good diagram? improving the cognitive effectiveness of diagrams in is development. In Knapp, Magyar, eds.: Intl Conf on Information Systems Development, Budapest, Hungary, Springer (August 31-2 2006)
12. Dullea, J., Song, I.Y., Lamprou, I.: An analysis of structural validity in entity-relationship modeling. Data Knowl. Eng. **47**(2) (2003) 167–205
13. Boufares, F., Bennaceur, H.: Consistency problems in er-schemas for database systems. Inf. Sci. **163**(4) (2004) 263–274
14. Damm, F.M., Hansen, B.S., Bruun, H.: On type checking in vdm and related consistency issues. In: VDM '91: Proceedings of the 4th International Symposium of VDM Europe on Formal Software Development-Volume I, London, UK, Springer-Verlag (1991) 45–62
15. Chen, P.P.S.: The entity-relationship model—toward a unified view of data. ACM Trans. Database Syst. **1**(1) (1976) 9–36
16. Hoxmeier, J.A.: Typology of database quality factors. Software Quality Journal **7**(3/4) (1998) 179–193
17. Kesh, S.: Evaluating the quality of entity relationship models. Information and Software Technology **37**(12) (dec 1995) 681–689
18. Briand, L.C., Morasca, S., Basili, V.R.: Property-based software engineering measurement. IEEE Trans. Softw. Eng. **22**(1) (1996) 68–86
19. Habra, N., Abran, A., Lopez Martin, M.A., Sellami, A.: A framework for the design and verification of software measurement methods. Journal of Systems and Software **81**(5) (2008) 633–648
20. Nelson, R.R., Todd, P.A., Wixom, B.H.: Antecedents of information and system quality: An empirical examination within the context of data warehousing. Journal of Management Information Systems **21**(4) (2005) 199–235
21. Lindland, O.I., Sindre, G., Solvberg, A.: Understanding quality in conceptual modeling. IEEE Softw. **11**(2) (1994) 42–49
22. Moody, D.L., Sindre, G., Brasethvik, T., Sølvberg, A.: Evaluating the quality of information models: empirical testing of a conceptual model quality framework. In: ICSE '03: Proceedings of the 25th International Conference on Software Engineering, Washington, DC, USA, IEEE Computer Society (2003) 295–305
23. Serrano, M., Trujillo, J., Calero, C., Piattini, M.: Metrics for data warehouse conceptual models understandability. Inf. Softw. Technol. **49**(8) (2007) 851–870

24. Manso, M.E., Genero, M., Piattini, M.: No-redundant metrics for uml class diagram structural complexity. In Eder, J., Missikoff, M., eds.: CAiSE. Volume 2681 of Lecture Notes in Computer Science., Springer (2003) 127–142
25. Genero, M., Piattini, M., Manso, M.E.: Finding "early" indicators of uml class diagrams understandability and modifiability. In: ISESE, IEEE Computer Society (2004) 207–216
26. Bansiya, J., Davis, C.G.: A hierarchical model for object-oriented design quality assessment. IEEE Trans. Software Eng. **28**(1) (2002) 4–17
27. Genero, M., Piattini, M., Manso, M.E., Garcia, F.: Early metrics for object oriented information systems. In: OOIS '00: Proceedings of the 6th International Conference on Object Oriented Information Systems, London, UK, Springer-Verlag (2000) 414–425
28. Cherfi, S.S.S., Akoka, J., Comyn-Wattiau, I.: Perceived vs. measured quality of conceptual schemas: An experimental comparison. In: ER (Tutorials, Posters, Panels & Industrial Contributions), Australian Computer Society (2007) 185–190
29. Piattini, M., Calero, C., Sahraoui, H., Lounis, H.: Object-relational database metrics (2003)
30. Mens, T., Demeyer, S., eds.: Software Evolution. Springer (2008)
31. Wahler, M.: Using Patterns to Develop Consistent Design Constraints. PhD thesis, ETH Zurich, Switzerland (2008)
32. Eick, C.F.: A methodology for the design and transformation of conceptual schemas. In: VLDB '91: Proceedings of the 17th International Conference on Very Large Data Bases, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. (1991) 25–34
33. Rauh, O., Stickel, E.: Standard transformations for the normalization of er schemata. In: CAiSE, Springer (1995) 313–326
34. Assenova, P., Johannesson, P.: Improving quality in conceptual modelling by the use of schema transformations. In: ER '96: Proceedings of the 15th International Conference on Conceptual Modeling, London, UK, Springer-Verlag (1996) 277–291
35. Hainaut, J.L.: The transformational approach to database engineering. In Lämmel, R., Saraiva, J., Visser, J., eds.: Generative and Transformational Techniques in Software Engineering. Volume 4143 of LNCS., Springer (2006) 95–143
36. OMG: UML 2.0 OCL Final Adopted Specification. OMG (2003)

Empirical Validation of Measures for UML Class Diagrams: A Meta-Analysis Study

M. Esperanza Manso[1], José A. Cruz-Lemus[2], Marcela Genero[2], Mario Piattini[2]

[1] GIRO Research Group, Department of Computer Science, University of Valladolid, Campus Miguel Delibes, E.T.I.C., 47011, Valladolid, Spain
manso@infor.uva.es

2ALARCOS Research Group, Department of Technologies and Information Systems, University of Castilla-La Mancha, Paseo de la Universidad, 4
13071 Ciudad Real, Spain
JoseAntonio.Cruz@uclm.es Marcela.Genero@uclm.e Mario.Piattini@uclm.es

Abstract. The main goal of this paper is to show the findings obtained through a meta-analysis study carried out with the data obtained from a family of controlled experiments. This consisted of 5 experiments performed in academic environments, which were carried out to validate empirically two hypotheses applied to UML class diagrams. These hypotheses investigate 1) The dependence between the structural complexity and size of UML class diagrams on one hand and their cognitive complexity on the other, as well as 2) The dependence between the cognitive complexity of UML class diagrams and their comprehensibility and modifiability. We carried out a meta-analysis, as it allows us to integrate the individual findings obtained from the execution of a family of experiments carried out to test the aforementioned hypotheses. The meta-analysis reveals that the measures related to associations and generalizations have a strong correlation with the cognitive complexity, and that the cognitive complexity has a greater correlation to comprehensibility than to modifiability. These results have implications from the points of view of both modeling and teaching, revealing which UML constructs are most influential when modelers have to comprehend and modify UML class diagrams. In addition, the measures related to associations and generalizations could be used to build prediction models.

Keywords: *meta-analysis, experiments, UML class diagrams, comprehensibility, modifiability, structural complexity, size.*

1. Introduction

The Model-Driven Development paradigm (MDD) [2] is an emerging approach for software development which is of ever-increasing interest to both the research community and software practitioners. MDD considers models as end-products rather than simply as means to produce software. The basic strategy in this approach is the use of

model transformations to obtain the final software product. In this context the quality focus has shifted from code to models, given that the quality of the models obtained through transformations is of great importance. This is because it will ultimately determine the quality of the software systems produced. Since, in the context of MDD, maintenance must be done on models, we are concerned about sub-characteristics of maintainability, such as the comprehensibility and modifiability of UML class diagrams. Class diagrams constitute the backbone of a system design and they must be comprehensible and flexible enough to allow the modifications that reflect changes in the things they model to be incorporated easily. We have based our work on the model shown in Figure 1 [1, 3]. This model constitutes a theoretical basis for the development of quantitative models relating to internal and external quality attributes and has been used as the basis for a great amount of empirical research into the area of structural properties of software artefacts [4-6]. In the study reported here, we have assumed a similar representation for UML class diagrams. We hypothesize that the structural properties (such as structural complexity and size) of a UML class diagram have an effect on its cognitive complexity. Cognitive complexity can be defined as the mental burden placed by the artefact on the people who have to deal with it (e.g. modellers, maintainers). High cognitive complexity will result in the production of an artefact that has reduced comprehensibility and modifiability, which will consequently affect its maintainability.

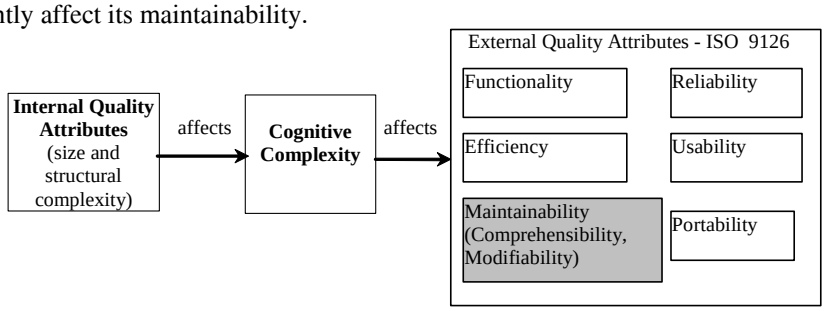


Figure 1. Relationship between structural properties, cognitive complexity, and external quality attributes, based on [1, 3]

The main motivation behind the research we have been carrying out is to validate this model, formulating two main hypotheses based on each of the arrows in Figure 1: 1) Size and structural complexity of UML class diagrams affect cognitive complexity, 2) Cognitive complexity affects the comprehensibility and modifiability of UML class diagrams. To measure the content of each box of Figure 1 we have defined some measures, which will be introduced in Section 3. In order to test such hypotheses, we carried out 5 experiments ,which constitute a family of experiments [7, 8].

The data analysis carried out in each individual experiment did not allow us to obtain conclusive results. This led us to carry out a meta-analysis study. Meta-analysis has been recognised as an appropriate way to aggregate or integrate the findings of empirical studies in order to build a solid body of knowledge on a topic based on empirical evidence [9-11]. Moreover, the need for meta-analysis is gaining relevance in empirical research, as is demonstrated by the fact that it is a recurrent topic in various forums related to Empirical Software Engineering. In other areas, such as psychology or medicine, a single study is extremely unlikely to be definitive. Dozens and even

hundreds of studies on the same topic may follow. In Empirical Software Engineering it is unusual for a large amount of studies concerning the same topic to take place, but it is necessary to cross the borders of individual studies so as to extract more global conclusions from families of experiments, with or without significant results. Meta-analysis is a tool for extracting these global conclusions from families of experiments, as it allows us to estimate the global effect size of the whole family, as well as to measure the accuracy of this measure and to evaluate the significance of effect size with respect to the hypotheses under study.

The main goal of the current paper is to present a meta-analysis study that would serve to integrate the results obtained from previous experimentation. In this way, meta-analysis contributes to the obtaining of a solid body of knowledge concerning the usefulness of the measures for UML Class diagrams.

The remainder of the paper is organised as follows: Section 2 presents the related work; Section 3 describes the family of experiments. The Meta-analysis study is presented in Section 4. Finally, the last section presents some concluding remarks and outlines our future work.

2. Related work

In empirical studies within the context of Empirical Software Engineering, special interest has been placed on external quality attributes such as maintainability, comprehensibility, modifiability, etc. Initially, the focus was on code or detailed design artifacts [12-16]. Later, given the increasing relevance of modeling, the focus shifted to models. The comprehension and modification of UML diagrams have been the goals of a great amount of the empirical research on UML diagrams carried out in recent years [17-25].

Our previous works address the influence of both the structural complexity and size of UML class diagrams on their comprehensibility and modifiability. A summary of these works is shown in [8]. In all of them, several controlled experiments were carried out, but the data analysis took place individually for each experiment, in some cases obtaining controversial results. For this reason and owing to the increasing need to investigate the UML constructs that have most influence on the comprehension and modification of UML class diagrams, we decided to integrate the results of homogenous experiments through a meta-analysis study, which is the main goal of the current work.

3. The Family of Experiments

Isolated studies (or experiments) hardly ever provide enough information to answer the questions posed in a research study [10, 26, 27]. Thus, it is important for experiments to be part of families of studies [26]. Common families of studies can contribute to devising important and relevant hypotheses that may not be suggested by individual experiments. More importantly, they allow researchers to answer questions that are beyond the scope of individual experiments, and to generalize findings across

studies. In this work we will comment on five experiments, whose main contextual characteristics are summarized in Table 1.

Table 1. Characteristics of the experiments

Study	#Subjects	University	Date	Year
E1	72	University of Seville (Spain)	March 2003	4th
R1	28		March 2003	
E2	38	Univ. of Castilla-La Mancha (Spain)	April 2003	3rd
R21	23	University of Sannio (Italy)	June 2003	4th
R22	71	University of Valladolid (Spain)	Sept. 2005	3rd

To perform the experiments, we followed the guidelines provided in [28, 29].

3.1 Planning of Experiments

In this sub-section we will define the common framework of all the studies:

1. **Preparation.** The family has a double goal, defined as:
   - Goal 1: To analyze the structural complexity of UML class diagrams with respect to their relationship with cognitive complexity from the viewpoint of software modelers or designers in an academic context.
   - Goal 2**:** To analyze the cognitive complexity of UML class diagrams with respect to their relationship with comprehensibility and modifiability from the viewpoint of software modelers or designers in an academic context.
2. **Context definition.** In these studies, we have used students as experimental subjects. The tasks to be performed did not require high levels of industrial experience, so we believed that these subjects might be considered appropriate, as is pointed out in several works [26, 30]. In addition, working with students implies a set of advantages, such as the facts that the students' prior knowledge is fairly homogeneous, a large number of subjects is readily available, and there is the possibility of testing experimental design and initial hypotheses [31]. A further advantage of using novices as subjects in experiments on understandability is that the cognitive complexity of the object under study is not hidden by the subjects' experience.
3. **Material.** The experimental materials consisted of a set of UML class diagrams suitable for the family goals. The selected UML class diagrams covered a wide range of the metrics values, considering three types of diagrams: Difficult to maintain (D), Easy to Maintain (E) and Moderately difficult to maintain (M). Some were specifically designed for the experiments and others were obtained from real applications. Each diagram had some documentation attached, containing, among other things, four comprehension and four modification tasks.

3.2. How the Individual Experiments were conducted.

We shall now explain the experimental plan of the different members of the family of experiments. The variables considered for measuring the structural complexity and size were the set of 11 measures presented in Table 7 in Appendix A. The *CompSub* measure is the subjective perception given by the subjects with regard to the complexity of the diagrams they have to work with during the experimental task. We consider *CompSub* to be a measure of cognitive complexity. The allowable values of this variable are: Very simple, Moderately simple, Average, Moderately complex and Very complex. To measure the comprehensibility and modifiability of UML class diagrams, we considered the time (in seconds) taken by each subject to complete the comprehensibility and modifiability tasks. We called these measures the *Comprehensibility* and *Modifiability* time.

We used a counter-balanced between-subjects design, i.e., each subject works with only one diagram. The diagrams were randomly assigned and each diagram is considered by the same number of subjects.

We formulated the following hypotheses, which are derived from the family's goals:

- $H_{0,1}$: The structural complexity and size of UML class diagrams are not correlated with the cognitive complexity. $H_{1,1}$: $\neg H_{0,1}$
- $H_{0,2}$: The cognitive complexity of UML class diagrams is not correlated with their comprehensibility and modifiability. $H_{1,2}$: $\neg H_{0,2}$

All the experiments were supervised and time-limited. More details can be found in [7, 8]. Finally, we used SPSS [32] to perform all the statistical analyses and the tool Comprehensive Meta Analysis [33] was employed to perform the meta-analysis.

3.3 Experiment 1 (E1) and Replication (R1)

On testing the hypotheses we obtained the following findings:

- **Correlation between structural and cognitive complexities (hypotheses 1 - goal 1).** The correlation between the *CompSub* variable and the 11 metrics was significant at a 0.05 level for E1. We also obtained a significant correlation for R1 in all cases, with the exception of the NM, NGen and MaxDIT metrics. Because of constraints on space, we do not include the coefficient tables here.
- **Correlation between cognitive complexity and comprehensibility and modifiability (hypotheses 2 - goal 2).** Table 2 indicates that for E1, the subjective complexity is significantly correlated to the comprehensibility. For R1, the results are not significant, and they are hence unfavourable to goal 2 of the family.

Table 2. Results related to goal 2 for E1 & R1

Variables correlated	E1 (n=62)		R1(n= 22)	
	$\rho_{spearman}$	p-value	$\rho_{spearman}$	p-value
CompSub vs *Comprehensibility*	0.266	0.037	0.348	0,111
CompSub vs *Modifiability*	0.132	0.306	0.270	0.217

The results obtained are now related to the family's goals:

- Goal 1: Structural and cognitive complexities present a positive significant correlation for all metrics, except for NM, NGen and MaxDIT in R1.
- Goal 2: Cognitive complexity seems to be positively correlated to the effort needed to comprehend UML class diagrams, but the results are significant only for E1. At the same time, there is no correlation with the effort needed to modify the diagrams. A possible explanation for this could be that the subjects base their perception on the difficulty of the first tasks that they perform, which in this case are the comprehension ones.

3.4 Experiment 2 (E2) and its Replications (R21 & R22)

In these studies, goals and variables are the same as in the previous ones, but the diagrams used were different, and context and design have also been improved. More detailed information about them can be found in [8].

Apart from the family's variables, some other variables have been added, in order to validate the results:

- CompCorrectness = # correct comprehension tasks / # total tasks performed
- CompCompleteness = # correct comprehension tasks / # total tasks to perform
- ModifCorrectness = # correct modification tasks / # total tasks performed
- ModifCompleteness = # correct modification tasks / # total tasks to perform

Again, we use a within-subjects design, but in this case it has been improved by blocking the subjects' experience. A pre-test was performed, the results of which led to the subjects' being divided into two groups. Each diagram was then assigned to the same number of subjects from each group. More details about this process can be found in [8].

The *Comprehensibility* and *Modifiability* measures were only included when the tasks performed had a minimum quality level, and it was for this reason that we used the newly introduced variables, presented previously. The subjects who attained under 75% in correctness and completeness were excluded from the study. In fact their exclusion improved the behaviour of the dependent variables, i.e, symmetry and outliers.

On testing the hypotheses we obtained the following findings:

- **Correlation between** structural **and cognitive complexities (hypothesis 1-goal 1).** Table 3 summarizes the metrics that are significantly correlated with the *CompSub* variable.

Table 3. Goal 1 results for E2, R21 & R22

Study	Significantly correlated metrics
E2	NC, NAssoc, NGen, NGenH, MaxDIT (5 out of 11)
R21	All except for NM, NGenH and MaxAgg (8 out of 11)
R22	All except for NM (10 out of 11)

- **Correlation between cognitive complexity and comprehensibility and modifiability (goal 2).** Table 4 indicates that for all the studies, the subjective complexity is significantly correlated to the comprehensibility. For modifiability, the results are not significant in all cases, and are once again unfavourable to goal 2 of the family.

Table 4. Results related to goal 2 for E2, R21 & R22

Variables correlated	E2			R21			R22		
	$\rho_{spearman}$	p-value	N	$\rho_{spearman}$	p-value	N	$\rho_{spearman}$	p-value	N
CompSub vs *Comprehensibility*	0.343	**0.049**	33	0.410	**0.065**	21	0.353	**0.003**	70
CompSub vs *Modifiability*	0.337	0.099	25	0.156	0.500		0.165	0.173	

These studies were based on two goals related to the connection of the different elements of the theoretical model which we used as a starting point. The results were:

- Goal 1: We have favourable results which admit a correlation between the structural and the cognitive complexities of UML class diagrams. Most of the metrics are significantly correlated with the subjective complexity in the different studies, especially those related to inheritance hierarchies.
- Goal 2: The results are also in favour of the hypothesis that relates cognitive complexity to the comprehensibility of UML class diagrams.

3.5 Threats to the Validity of the Family of Experiments.

The main threats to the validity of the family are the following:

- **Conclusions validity.** The number of subjects in R1, E2 and R21 is quite low, and subjects were selected by convenience. Our conclusions must therefore be applied to the population represented by the subjects.
- **Internal validity.** We have found correlation between the variables, which implies the possibility of the existence of that causality, but not the causality itself. Moreover, R21 materials were written in English, which is not the mother language of the subjects (Italians). This fact may have increased the times taken to perform the tasks, especially those of modification.
- **External validity.** It would be advisable to perform some replications with data extracted from real projects, in an effort to generalise the results obtained.

4. Meta-Analysis Study

There are several statistical methods that allow us to accumulate and interpret a set of results obtained through different inter-related experiments, since they check similar hypotheses [34-38]. The limitations of all these methods are commented upon in [11]. There are three main ways in which to perform this process: meta-analysis, significance level combination and vote counting.

The first and second are those most commonly used in Software Engineering, and it is for this reason that we comment upon them now:

- Meta-analysis is a set of statistical techniques that allow us to combine the different effect size measures (or treatment effect) of the individual experiments. There are several metrics to obtain this value, e.g. the means difference and the correlation coefficients, among others [35]. The objective is to obtain a global effect, the treatment effect of all experiments. As effect size measures may come from different environments and may even not be homogeneous, it is necessary to obtain a standardized measure of each one. For example, the dependence between two variables could be measured by different coefficients or scales The global effect size is obtained as a weighted average of standardized measures, in which the most commonly used weights are the sample size or the standard deviation. Together with the estimation of the global effect size, we can provide an estimated confidence interval and a p-value which allows us to decide on the meta-analysis hypotheses. We can find several applications of this technique in Empirical Software Engineering in the following works ,among others:
  - Miller and McDonald [39] synthesize the results of a study on the effect that a tool had on the effectiveness and efficiency of inspections. The effect sizes were obtained from the correlation coefficients.
  - Dybå et al. [40] perform a meta-analysis for studying the effect of pair-programming on quality, duration and effort. The effect size is measured with a mean difference. This technique was also used in [41-43] to obtain conclusions about experiments which evaluated different inspection techniques.
- Signification level combination. A mean, or another statistic is obtained, in order to summarize the different signification levels (α) of the experiments. An application of this technique can be found in [41], in which certain inspection techniques are studied in an experiment and four replications.

In the present study, we have a family of experiments whose main goals are:

1. To study the influence of metrics on the cognitive complexity of UML class diagrams.
2. To study the influence of cognitive complexity on the comprehensibility and modifiability of UML class diagrams.

The use of meta-analysis will allow us to extract global conclusions, despite the fact that some of the experimental conditions are not the same. As we have mentioned previously, we will need to standardize the effect sizes. In this meta-analysis we used correlation coefficients (r_i) that, once transformed (Fisher transformation), provide the effect sizes that have a Normal distribution (z_i), what makes them easier to use. The

global effect size is obtained using the Hedges' g metric [35, 44], that is a weighted mean which has the proportional weights to the experiment size (equation 1).

$$\overline{Z} = \frac{\sum_i w_i z_i}{\sum_i w_i} \qquad W_i = 1/(n_i\text{-}3) \tag{1}$$

The higher the value of Hedges' g is, the higher the corresponding correlation coefficient is too. For studies in Software Engineering, we can classify effect sizes into small, medium and large [44]. We rely on the use of the five empirical studies, previously presented in this work, which means that the conclusions about our goals will be extracted from five different results.

4.1. Meta-analysis Results

Firstly, a meta-analysis for each metric-*CompSub* pair will be carried out, taking into account the fact that the hypothesis test is one-tailed, i.e., we consider as null-hypothesis that the correlation is now above zero. In Table 5 we present the global estimation of the correlation coefficient, a confidence interval at 95%, the p-value and the value for Hedges' g, including a classification of the effect size as large (L), medium (M) or small (S).

Table 5. Meta-analysis of metrics-*CompSub*

H0: $\rho \leq 0$	Correlation (ρ) Global effect size	Lower limit	Upper limit	p-value	Hedges'g
NC	0.566	0.464	0.653	0.0000	1.322(L)
NA	0.541	0.435	0.632	0.000	1.219(L)
NM	0.177	0.040	0.307	0.012	0.339(S)
Nassoc	0.566	0.465	0.653	0.000	1.318(L)
NAgg	0.481	0.368	0.581	0.000	1.051(M)
NDep	0.484	0.371	0.584	0.000	1.060(M)
NGen	0.484	0.371	0.584	0.000	1.018 (L)
NGenH	0.422	0.302	0.529	0.000	0.903 (M)
NAggH	0.393	0.270	0.504	0.000	0.814 (M)
MaxDIT	0.492	0.379	0.590	0.000	1.080 (L)
MaxHAgg	0.360	0.233	0.474	0.000	0.734 (M)

The results observed are in favour of the existence of a positive correlation between cognitive complexity and the 11 metrics that measure the structural complexity

and size of UML class diagrams. In fact, most of the effect sizes are medium or large, with the exception of NM, which is small. The size metrics that have most influence upon the cognitive complexity are NC and NA, while the complexity metrics that have most influence upon cognitive complexity are related to aggregations (NAgg) and generalizations (NGen and MaxDIT). We can conclude that those diagrams with many classes and attributes will have an increased cognitive complexity. Moreover, class diagram models using many inheritance and aggregation mechanisms will also have an increased cognitive complexity.

With regard to the hypotheses derived from goal 2, Table 6 shows that we can admit the existence of correlation between the cognitive complexity and the two measures, *Comprehensibility* and *Modifiability*, which measure quality attributes of UML class diagrams.

Table 6. Meta-analysis of *CompSub-Comprehensibility* and *Modifiability* time

H0: $\rho \leq 0$	**Correlation (ρ) gobal effect size**	**Lower limit**	**Upper limit**	**p-value**	**Hedges'g**
Comprehensibility Time	0.330	0.200	0.449	0.000	0.684 (M)
Modifiability Time	0.186	0.044	0.320	0.011	0.368(M)

The effect sizes are medium in both cases, but the correlation estimation of *Comprehensibility* is larger than the correlation of *Modifiability*. So we can conclude that, the more cognitive complexity a diagram contains, the more difficult it will be to comprehend and modify.

As an example, Figure 2 presents in diagram form the meta-analysis of the relationship of a couple of metrics and the *CompSub* measure, and the relationship between their comprehensibility and cognitive complexity.

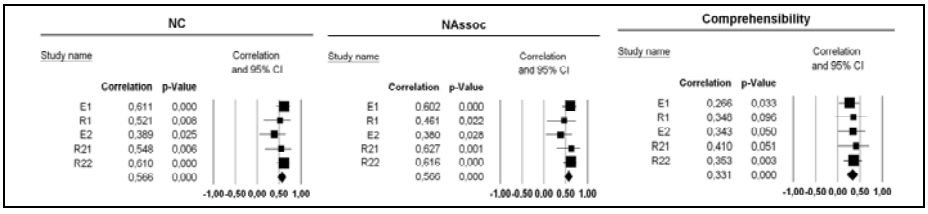


Figure 2. Meta-analysis for NC-*CompSub*, NAssoc-*CompSub* and *CompSub-Comprehensibility*

5. Conclusions

The main goal of this work has been that of validating a theoretical model which relates the structural complexity and size of UML class diagrams and cognitive complexity to two of their external quality attributes: comprehensibility and modifiability

(Figure 1). For that purpose, we carried out a meta-analysis study with the data obtained from a family of five experiments. The meta-analysis results are in favour of the model under inspection with regard to the two goals being pursued:

- Goal 1: structural complexity is correlated with cognitive complexity, especially with that related to associations and generalizations. An increase in the number of classes and attributes within classes also increases the cognitive complexity of UML class diagrams.
- Goal 2: cognitive complexity influences both the comprehensibility time and modifiability time of UML class diagrams, but this is especially true in the former case.

These results are relevant, as they point to a means of controlling the level of certain quality attributes of UML class diagrams from the modelling phase. The findings also have implications, both practically and in terms of teaching, providing information about which UML constructs may have more implications in the effort to understand and maintain UML class diagrams. When alternative designs of UML class diagrams exist, it could be advisable to select the one which minimizes these constructs.

Moreover, the measures related to associations and generalizations could be used to build prediction models, to evaluate how the time taken to understand or modify an UML class diagram increases; we have done this prediction modelling in [8]. In future work we plan to refine the prediction models obtained, using the data obtained in the whole family of experiments.

Further work is needed to confirm the findings of the current study, improving different issues:

1. Increasing the class diagram sample, as the results generalization depends on the sample of objects examined with respect to the whole object population. This meta-analysis covers 33 (24+9) UML class diagrams.
2. Working with subjects from different fields, preferably real practitioners or students from other universities, in the quest to generalize the results with regard to the subject population.
3. Improving the modifying tasks, to make them as real as possible.
4. Investigating other metrics to do with cognitive complexity.

Also pending is the carrying out of a similar study with the measures we have defined for UML statechart diagrams [45] and OCL expressions [46]

Acknowledgements

This research is part of the ESFINGE project (TIN2006-15175-C05-05) financed by the “Ministerio de Educación y Ciencia (Spain)”, the IDONEO project (PAC08-0160-6141) financed by “Consejería de Ciencia y Tecnología de la Junta de Comunidades de Castilla-La Mancha”.

References

1. L. Briand, S. Morasca and V. Basili, "Defining and Validating Measures for Object-Based High-Level Design", *IEEE Transactions on Software Engineering*, 25, 1999, pp. 722-743.
2. C. Atkinson and T. Kühne, "Model Driven Development: a Metamodeling Foundation", *IEEE Software*, 20, 2003, pp. 36-41.
3. ISO-IEC, "ISO/IEC 9126. Information Technology - Software Product Quality", 2001.
4. K. El-Emam and W. Melo, "The Prediction of Faulty Classes Using Object-Oriented Design Metrics", National Research Council of Canada, NRC/ERB1064, 1999.
5. K. El-Emam, S. Benlarbi, N. Goel and S. Rai, "The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics", *IEEE Transactions on Software Engineering*, 27(7), 2001, pp. 630-650.
6. G. Poels and G. Dedene, "Measures for Assessing Dynamic Complexity Aspects of Object-Oriented Conceptual Schemes", *19th International Conference on Conceptual Modelling (ER 2000)*, 2000, pp. 499-512.
7. M. Genero, M. E. Manso and M. Piattini, "Early Indicators of UML Class Diagrams Understandability and Modifiability", *ACM-IEEE International Symposium on Empirical Software Engineering*, 2004, pp. 207-216.
8. M. Genero, M. E. Manso, A. Visaggio, G. Canfora and M. Piattini, "Building Measure-Based Prediction Models for UML Class Diagram Maintainability", *Empirical Software Engineering*, 12, 2007, pp. 517-549.
9. M. Lipsey and D. Wison, *Practical Meta-Analysis*, Sage, 2001.
10. J. Miller, "Applying Meta-Analytical Procedures to Software Engineering Experiments", *Journal of Systems and Software*, 54, 2000, pp. 29-39.
11. L. M. Pickard, "Combining Empirical Results in Software Engineering", University of Keele, T-R V1, 2004.
12. W. Li and S. Henry, "Object-Oriented Metrics that Predict Maintainability", *Journal of Systems and Software*, 23, 1993, pp. 111-122.
13. R. Harrison, S. Counsell and R. Nithi, "Experimental Assessment of the Effect of Inheritance on the Maintainability of Object-Oriented Systems", *Journal of Systems and Software*, 52, 2000, pp. 173-179.
14. F. Fioravanti and P. Nesi, "Estimation and Prediction Metrics for Adaptative Maintenance Effort of Object-Oriented Systems", *IEEE Transactions on Software Engineering*, 27(12), 2001, pp. 1062-1083.
15. L. Briand, C. Bunse and J. Daly, "A Controlled Experiment for Evaluating Quality Guidelines on the Maintainability of Object-Oriented Designs", *IEEE Transactions on Software Engineering*, 27(6), 2001, pp. 513-530.
16. E. Arisholm and D. I. K. Sjøberg, "Evaluating the Effect of a Delegated versus Centralized Control Style on the Maintainability of Object-Oriented Software", *IEEE Transactions on Software Engineering*, 30(8), 2004, pp. 521-534.
17. M. C. Otero and J. J. Dolado, "Evaluation of the Comprehension of the Dynamic Modeling in UML", *Information and Software Technology*, 46(1), 2004, pp. 35-53.
18. M. C. Otero and J. J. Dolado, "An Empirical Comparison of the Dynamic Modeling in OML and UML", *Journal of Systems and Software*, 77(2), 2005, pp. 91-102.

19. J. A. Cruz-Lemus, M. Genero, M. E. Manso and M. Piattini, "Evaluating the Effect of Composite States on the Understandability of UML Statechart Diagrams", *8th International Conference on Model-Driven Engineering, Languages and Systems (MoDELS 2005)*, 2005, pp. 113-125.
20. H. C. Purchase, L. Colpoys, M. McGill, D. Carrington and C. Britton, "UML Class Diagram Syntax: an Empirical Study of Comprehension", *Australian Symposium on Information Visualisation*, 2001, pp. 113-120.
21. H. C. Purchase, L. Colpoys, M. McGill and D. Carrington, "UML Collaboration Diagram Syntax: an Empirical Study of Comprehension", *1st International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT'02)*, 2002, pp. 13-22.
22. M. Staron, L. Kuzniarz and C. Wohlin, "Empirical Assessment of Using Stereotypes to Improve Comprehension of UML Models: a Set of Experiments", *Journal of Systems and Software*, 79, 2006, pp. 727-742.
23. S. Xie, E. Kraemer and R. E. K. Stirewalt, "Empirical Evaluation of a UML Sequence Diagram with Adornments to Support Understanding of Thread Interactions", *15th IEEE International Conference on Program Comprehension (ICPC'07)*, 2007, pp. 123-134.
24. S. Yusuf, H. Kagdi and J. I. Maletic, "Assessing the Comprehension of UML Class Diagrams via Eye Tracking", *15th IEEE International Conference on Program Comprehension (ICPC'07)*, 2007, pp. 113-122.
25. C. F. J. Lange and M. R. V. Chaudron, "Interactive Views to Improve the Comprehension of UML Models - An Experimental Validation", *15th IEEE International Conference on Program Comprehension (ICPC'07)*, 2007, pp. 221-230.
26. V. Basili, F. Shull and F. Lanubile, "Building Knowledge through Families of Experiments", *IEEE Transactions on Software Engineering*, 25, 1999, pp. 456-473.
27. F Shull, J. Carver, G. Travassos, J. Maldodano, R. Conradi and V. Basili, "Replicated Studies: Building a Body of Knowledge about Software Reading Techniques", in *Lecture Notes on Empirical Software Engineering*, World Scientific, 2003.
28. C. Wohlin, P. Runeson, M. Hast, M. C. Ohlsson, B. Regnell and A. Wesslen, *Experimentation in Software Engineering: an Introduction.* Kluwer Academic Publisher, 2000.
29. N. Juristo and A. Moreno, *Basics of Software Engineering Experimentation*, Kluwer Academic Publishers, 2001.
30. M. Höst, B. Regnell and C. Wohlin, "Using Students as Subjects - a Comparative Study of Students & Professionals in Lead-Time Impact Assessment", *4th Conference on Empirical Assessment & Evaluation in Software Engineering (EASE 2000)*, 2000, pp. 201-214.
31. D. I. K. Sjoberg, J. E. Hannay, O. Hansen, V. Kampenes, A. Karahasanovic, N. K. Liborg and A. C. Rekdal, "A Survey of Controlled Experiments in Software Engineering", *IEEE Transactions on Software Engineering*, 31(9), 2005, pp. 733-753.
32. SPSS, SPSS 12.0, Syntax Reference Guide, 2003,
33. Biostat, Comprehensive Meta-Analysis v2, 2006, http://www.meta-analysis.com/
34. G. V. Glass, B. McGaw and M. L. Smith, *Meta-Analysis in Social Research*, Sage Publications, 1981.

35. L. V. Hedges and I. Olkin, *Statistical Methods for Meta-Analysis*, Academia Press, 1985.
36. R. Rosenthal, *Meta-Analytic Procedures for Social Research*, Sage Publications, 1986.
37. J. A. Sutton, R. K. Abrams, R. D. Jones, A. T. Sheldon and F. Song, *Methods for Meta-Analysis in Medical Research*, John-Wiley & Sons, 2001.
38. F. M. Wolf, *Meta-Analysis: Quantitative Methods for Research Synthesis*, Sage Publications, 1986.
39. J. Miller and F. McDonald, "Statistical Analysis of Two Experimental Studies", University of Strathclyde, EFoCS-31-98, 1998.
40. T. Dybå, E. Arisholm, D. I. K. Sjøberg, J. E. Hannay and F. Shull, "Are Two Heads Better than One? On the Effectiveness of Pair Programming", *IEEE Software*, 24(6), 2007, pp. 10-13.
41. W. Hayes, "Research in Software Engineering: a Case for Meta-Analysis", *6th IEEE International Symposium on Software Metrics (METRICS'99)*, 1999, pp. 143-151.
42. O. Laitenberger, K. El-Emam and T. Harbich, "An Internally Replicated Quasy-Experimental Comparison of Checklist and Perspective-based Reading of Code Documents", IESE, 006.99/e, 1999.
43. A. Porter and M. Johnson, "Assessing Software Review Measurement: Necessary and Sufficient Properties for Software Measures", *Information and Software Technology*, 42(1), 1997, pp. 35-46.
44. V. Kampenes, T. Dybå, J. E. Hannay and D. I. K. Sjoberg, "A Systematic Review of Effect Size in Software Engineering Experiments." *Information and Software Technology*, 49(11-12), 2007, pp. 1073-1086.
45. J. A. Cruz-Lemus, M. Genero and M. Piattini, "Metrics for UML Statechart Diagrams", in *Metrics for Software Conceptual Models*, Imperial College Press, UK., 2005.
46. L. Reynoso, M. Genero and M. Piattini, "Measuring OCL Expressions: An approach based on Cognitive Techniques", in *Metrics for Software Conceptual Models*, Imperial College Press, UK., 2005.
47. M. Genero, M. Piattini and C. Calero, "Early Measures for UML Class Diagrams", *L'Object*, 6(4), 2000, pp. 495-515.
48. M. Genero, G. Poels, M. E. Manso and M. Piattini, "Defining and Validating Metrics for UML Class Diagrams", in *Metrics for Software Conceptual Models*, Imperial College Press, UK., 2005.

Appendix A

After studying the UML metamodel, and having reviewed the literature concerning existing measures, we proposed a set of eight measures for the structural complexity of UML class diagrams [47, 48] (see Table 7). The proposed measures are related to the usage of UML relationships, such as associations, dependencies, aggregations and generalizations. In the study reported in this work, we have also considered traditional OO measures, such as size measures (see Table 7).

Table 7. Measures for UML class diagrams

	Measure Name	Measure definition
Size measures	Number of Classes (NC)	The total number of classes in a class diagram.
	Number of Attributes (NA)	The number of attributes defined across all classes in a class diagram (not including inherited attributes or attributes defined within methods). This includes attributes defined at class and instance level.
	Number of Methods (NM)	The total number of methods defined across all classes in a class diagram, not including inherited methods (as this would lead to double counting). This includes methods defined at class and instance level.
Structural complexity measures	Number of Associations (NAssoc)	The total number of association relationships in a class diagram.
	Number of Aggregations (NAgg)	The total number of aggregation relationships (each "whole-part" pair in an aggregation relationship).
	Number of Dependencies (NDep)	The total number of dependency relationships.
	Number of Generalizations (NGen)	The total number of generalization relationships (each "parent-child" pair in a generalization relationship).
	Number of Generalization Hierarchies (NGenH)	The total number of generalization hierarchies, i.e. it counts the total number of structures with generalization relationships.
	Number of Aggregation Hierarchies (NAggH)	The total number of aggregation hierarchies, i.e. it counts the total numbers of "whole-part" structures within a class diagram.
	Maximum DIT (MaxDIT).	The maximum DIT value obtained for each class of the class diagram. The DIT value for a class within a generalization hierarchy is the longest path from the class to the root of the hierarchy (Chidamber and Kemerer, 1994).
	Maximum HAgg (MaxHAgg)	The maximum HAgg value obtained for each class of the class diagram. The HAgg value for a class within an aggregation hierarchy is the longest path from the class to the leaves.

Towards a Tool-Supported Quality Model for Model-Driven Engineering

Parastoo Mohagheghi, Vegard Dehlen, Tor Neple

SINTEF, P.O.Box 124 Blindern
N-0314 Oslo, Norway
{Parastoo.Mohagheghi, Vegard.Dehlen, Tor.Neple}@sintef.no

Abstract. This paper reviews definitions of model quality before introducing six properties of models that are important for building high-quality models. These are identified to be correctness, completeness, consistency, comprehensibility, confinement and changeability. We have earlier defined a quality model that separates intangible quality goals from tangible quality-carrying properties and practices that should be in place to support these properties. A part of that work was to define a metamodel for developing quality models with MDE in mind. In this paper we analyze existing literature in order to extract model quality properties and to build a quality model with focus on the quality of models. For this purpose the metamodel is implemented in a tool that allows us to model quality models. The advantage of defining the metamodel is learning how to precisely define quality elements and relations in the quality model, and building models that may be used to generate documentation, guidelines or checklists. The disadvantage is mainly in the research phase where the metamodel is not stable and undergoes changes.

Keywords: Quality model, model-driven engineering, metamodel, model quality.

1 Introduction

Model-Driven Software Development (MDSD) or Model-Driven Engineering (MDE) is an approach to software development that emphasizes using models when specifying, developing, analyzing, verifying and managing software systems[1]. Since MDE requires a model-centric software development approach, researchers have also worked on specific quality issues in MDE such as identifying characteristics of models that are important depending on the modeling purpose and how to achieve and evaluate them.

The work described in this paper also aims at identifying quality attributes and approaches to improve the quality of models. We have earlier defined a quality model with concepts and their relations to be able to build quality models for different domains and different targets of quality; for example models, languages or

[1] We use the term MDE in the remainder of this paper to cover these approaches.

transformations. This work is explained in [1] and [2]. We have also developed a prototype tool in Eclipse built on the metamodel that is described in [2]. This paper builds an instance of the quality model with focus on the quality of models. The goal is two-folded: 1) identifying quality-carrying properties of models and relating them to practices needed to achieve them; 2) presenting the tool and metamodel to the audience of this workshop to get feedback.

The remainder of this paper is organized as follows. Section 2 discusses concepts important for discussing quality of models and provides a classification of model quality-carrying properties. Section 3 introduces the metamodel and the prototype tool while Section 4 presents a model on the quality of models which is developed by using the concepts presented in previous sections, an extensive literature search and by using the tool. Section 5 is discussion and conclusion.

2 Model Quality

Discussing approaches to improve the quality of models is not possible without discussing what model quality means. Software quality in general has been subject of extensive research, as reflected in the various quality models and standards with their definitions of quality. We have discussed some existing quality models in [2]. Here we just provide a short description of our quality model and quality properties identified for models.

2.1 A Quality Model MDE

In [1] and [2], we discussed the need for defining a quality model[2] in MDE as a means to integrate quality work related to MDE. Our approach is built on the Dromey's quality model, which has three main principles: high-level quality attributes or *quality goals*, product properties that are important for achieving quality goals (called *quality-carrying properties*), and links between quality-carrying properties and quality goals [3]. Dromey's focus is on component-based systems and in order to establish the links, Dromey has identified four properties of components that impact software quality. These are:

- Correctness properties; related to the deployment of components and that rules are not violated; either internally or associated with their use in the context;
- Internal properties; how well a component is deployed according to its intended use or requirements, covering both correctness and other properties;
- Contextual properties; how to compose components in a context;
- Descriptive properties; requirements, design, implementations and user interfaces must be easy to understand and use for their intended purpose.

Although Dromey's model emphasizes separating intangible quality goals from tangible quality-carrying properties, it does not focus on how to construct these

[2] We called this a quality framework in earlier publications.

properties in a product. Therefore we have added the concept of *practice* to our quality model that helps achieving a quality-carrying property and maps to the concept of "means" in the work of Lindland et al. [4]. Adapting the approach to MDE requires identifying properties of models that impact software quality; equivalent to the four properties identified by Dromey for components. We discuss these properties in the next section.

2.2 Model Quality Goals and Properties

UML 1.5 defines a model as "an abstraction of a physical system with a certain purpose" [8]. Daniels defines three kinds of models based on their purposes:

- *Conceptual models* describe a situation of interest in the world, such as a business operation or factory process;
- *Specification models* define what a software system must do, the information it must hold, and the behavior it must exhibit. They assume an ideal computing platform;
- *Implementation models* describe how the software is implemented, considering all the computing environment's constraints and limitations.

MDA separates between CIM (Computational Independent Model), PIM (Platform Independent Model) and PSM (Platform Specific Model). Comparing with the Daniels definitions, a CIM is a conceptual model of the domain, sometimes called a domain model, and a vocabulary that is familiar to the practitioners of the domain is used in its specification. Thus it should be evaluated for its understandability by users and validity and completeness related to the domain ontology. PIMs may be both specification of what a system does and a platform-independent solution, while PSMs are clearly implementation models and are in the solution domain. PIMs and PSMs should be evaluated for other characteristics such as their correctness and consistency with the concepts in the CIM model. Other models that may be evaluated are pattern models, transformation models or even metamodels as models of models.

Various models may be used for communication, implementation, generation and execution, or documentation. Thus the users are either human beings (developers, customers, business analysts etc.) or tools that should interpret the models. There are some definitions of quality goals for various types of models and various purposes of modeling that we refer to here. We cannot provide an extensive review of model types and quality goals due to the limited space of this paper but refer only to the related work that is used as basis for our definitions.

One of the earliest and widely referred works on model quality goals is the article by Lindland et al. [4]. They refer to earlier work on the quality of requirement models that contain quality goals such as:

- Appropriate: specifications should be free of implementation concerns, concepts must be suitable for the domain and the systems role in the environment such as data processing.
- Conceptually clean: covers notions such as simplicity, clarity and ease of understanding.

- Complete: contain all needed information.
- Expressive economy: least number of statements.
- Unambiguous: every requirement has only one interpretation.
- Consistent: Not in conflict with one another.

We realize that the above goals are applicable to other types of models as well. There are also quality goals that are especially relevant for requirement models such as being verifiable, having a traceable identification of requirements and being testable.

Lindland et al. have defined a framework for the quality of conceptual models which relates model quality to modeling language, domain and the audience interpreting the models. They have further borrowed three linguistic concepts to classify quality goals of models. These quality types are:

- *Syntax.* Relates the model to the modeling language by describing relations among language constructs without considering their meaning. Syntactic quality is therefore how well the model corresponds to the language. Syntactic errors happen if a model contains symbols not defined in the language or a model lacks constructs or information to obey the language's grammar. Having a formal syntax helps prevention and detection of syntactic errors.
- *Semantics*. Relates the model to the domain by considering not only syntax but also relations among statements and their meaning. Semantic quality is how well the model corresponds to the domain or the knowledge of people from the domain. There are two semantic goals: validity and completeness. Validity means that all statements made by the model are correct and relevant to the problem. Completeness means that the model contains all the statements about the domain that are correct and relevant. The authors write that semantic quality is difficult to achieve and it is best to relax the requirements and agree therefore on feasible validity and completeness. Consistency, unambiguity and being minimal are also types of semantic quality that are covered if models are valid and complete. Semantic means (or practices as we called them) are adding or removing statements to make a model complete or valid, and consistency checking.
- *Pragmatics*. Relates the model to the audience's interpretation of it. There is one goal of pragmatic quality: comprehension. The comprehension requirement may be relaxed since it takes a lot of effort to develop a large model where every part is comprehensible by everyone. Pragmatic means are those that make a model easier to understand; for example inspection, visualization (use of graphical models instead of textual), filtering (hiding details, using different languages for different parts, and we may also add the MDA concept of models at different abstraction layers), and executable models that can be simulated to help understanding.

Fig.1 shows the quality goals related to each of the quality types and means identified to achieve them.

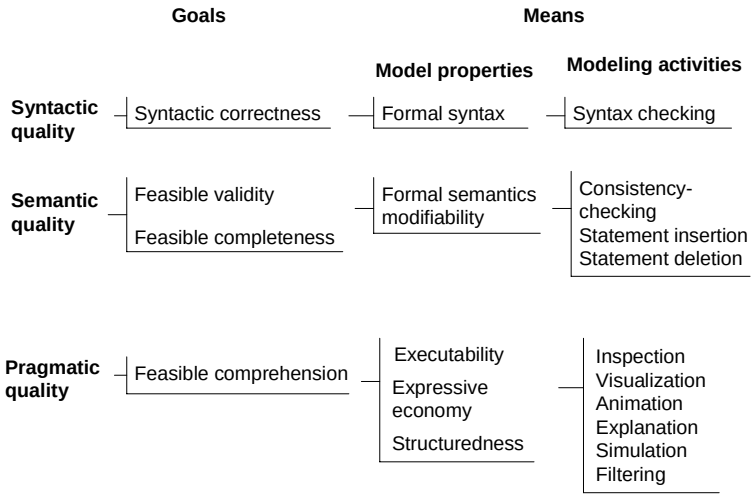


Fig. 1. Proposed quality goals and means in the framework of Lindland et al. [4]

This framework is later extended by others, among them Krogstie [5] who has added other quality goals to the framework such as *organizational quality* (defined as whether a model fulfils the goals of modeling and that all the goals of modeling are addressed through the model) and *technical pragmatic quality* defined as being interpretable by tools. Solheim and Neple have added *productivity* to the organizational goals and have defined two other quality goals relevant for MDE; i.e., transformability and maintainability [12].

Unhelkar has also used Lindland et al.'s framework but replaces the pragmatic quality with *aesthetics* [6]. He therefore defines three quality goals for UML models:

- *Syntax* with focus on correctness, for example does classes have correct attributes and operations, does attributes have correct types, etc. His definition of syntax also covers documentation, packaging and other issues related to understandability of models that Lindland et al. defined as pragmatic quality.
- *Semantics* or meaning with focus on completeness, consistency and representing the domain: does elements represent entities as they should, are dependencies correct, are models consistent with one another? Inspection of models is often the best way to check semantic quality since not all models are executable.
- *Aesthetics* with focus on symmetry and consistency: does a class have too many responsibilities, how many actors and use cases are shown in one diagram, how many activity diagrams are associated with a use case etc. These checks are both to improve the look and to help understanding.

His classification is different from Lindland et al., and we chose to use the definitions of Lindland et al. when classifying quality goals in the remainder of the paper.

Haesen writes that the boundary between syntax and semantics is sometimes blurred [10]. For example in [13], Harel and Rumpe state that as soon as a constraint can be automatically checked, it is a syntactic constraint, while a semantic constraint is formulated in natural language and cannot be checked automatically. This implies that if a modeling language has been well-formalized, more constraints can be automatically checked and are considered to be syntax, whereas for a badly formalized language, almost everything becomes semantic checking. Another view on syntax versus semantics is that syntax refers to the well-formedness of a single diagram or view, whereas semantics refers to the existence of a (mathematical) model defining the ”meaning” of a diagram and allowing reasoning and inference on specifications. Therefore although we keep the notions of syntax, semantics and pragmatics, we realize that we need a more precise definition of quality goals for models that is based on the experience of developers who rather talk of properties such as consistency or correctness.

Based on the above discussions and the results of an extensive literature search that we have done (some results are already published in [1] and [2], while we work on publishing the others), we have identified five basic quality properties of models that are shared between various models and are widely used in literature. These are important for achieving several quality goals such as maintainability of models and reliability of the generated software. They are also useful when discussing the impact of approaches to improve the quality of models. Others call them quality goals for models. The selection of terminology depends on the vocabulary used. These five quality properties of models are:

- *Correctness*; as including correct elements and correct relations between them and not violating rules and conventions; for example adhering to language syntax, style rules, naming rules or other conventions. Thus it covers both syntactic correctness (right syntax or well-formedness) and semantic correctness (right meaning and relations relative to the knowledge about the domain).
- *Completeness*; as having all the necessary information and being detailed enough; according to the goals of modeling. Completeness is a semantic quality.
- *Consistency*; as no contradictions in the models, related to semantic quality. It covers consistency between views that belong to the same level of abstraction or development phase (horizontal consistency), and between views that model the same aspect, but at different levels of abstraction or in different development phases (vertical consistency). The model should also be unambiguous; i.e. not allowing multiple interpretations.
- *Comprehensibility*; as being understandable by the intended users, related to the pragmatic quality. For human users, several aspects impact comprehensibility such as aesthetics of diagrams, organization of a model, model simplicity (or complexity which may be reduced by for example generalization and refactoring), conciseness (expressing much with little), and using domain concepts (Mitchell et al. call this *continuity* defined as carrying the structure and behavior of a problem domain into system and design models [7]). For tools,

having a formal syntax and semantic helps analysis and generation. Krogstie calls comprehensibility by human users as social pragmatic quality and comprehensibility by tools (or interpretability) as technical pragmatic quality.

- *Confinement* ; as being in agreement with the purpose of modeling and the type of system, and being restricted to the modeling goals, such as being at the right abstraction level and not having information not required (for example including implementation details in analysis models). Confinement is related to semantic quality.

A sixth property which has not received fair attention by literature so far but is of importance in a dynamic world is the *changeability* of models when the domain or our understanding of it changes or the solution must evolve because of changing requirements. We call these the *C6 properties* and think that they are fundamentally important in MDE and in any model-centric development process. Fig. 2 shows the quality properties in the transition from real world to software. The figure is inspired from a figure in [11] but has added new elements and quality properties to it.

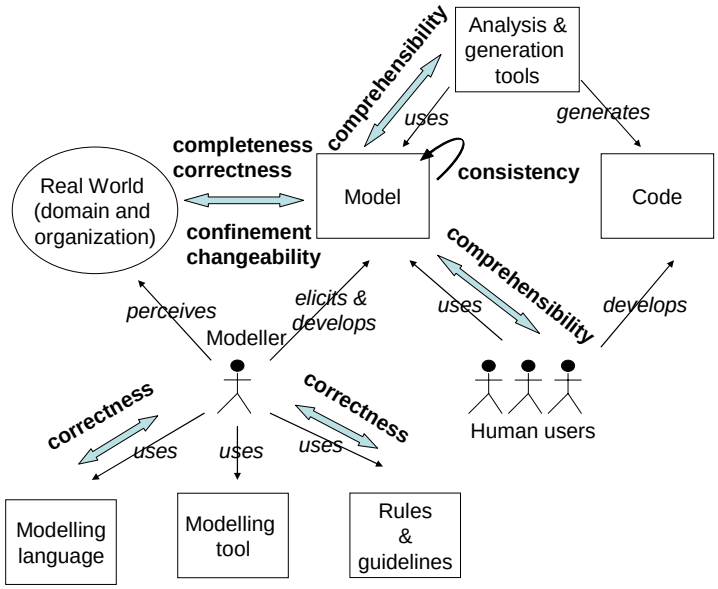


Fig. 2. Model-centric development with transformation of real world to running software

A model is a representation of a system and should be complete relative to the system it wants to represent and according to the modeling goals defined by the organization. The elements in the model should also be relevant to the domain and have correct relations relative to the knowledge we have of the domain. All these properties depend on the perception of the model developer from the domain and the modeling goals. Properties that are related to the hard assets (hard assets are shown with rectangles in Fig. 2, such as modeling language) may be evaluated automatically,

while those that are related to the soft assets (real world and human users) depend on perceptions; i.e., how a developer perceives the real world and how human users interpret the model. These require other means of evaluation such as experiments and inspections. We may also add reverse engineering to the figure: tools may generate models from the code. But this is not relevant for the discussion here.

The order of the first five properties is important when evaluating them: correctness should be in place before one can evaluate completeness, and both are necessary for consistency. Comprehensibility requires correct, complete and consistent models. Finally it is difficult to match a model against the real world to evaluate confinement if it is not comprehensible. Changeability presents another dimension and is required, otherwise the models cannot evolve with changes in the real world and get outdated.

We think that other quality goals of models can be defined as a combination of these properties. For example, maintainability requires almost all of the above properties, while reusability requires models that are comprehensible and changeable such as being well-organized. Other properties may be added if we discover the need since we want a combination of properties that addresses the following requirements: 1) completeness, as including all basic properties, 2) parsimony or being minimal, meaning that all of the properties are necessary, and 3) independence or orthogonality of the properties [9]. All of the above properties can be achieved by practices and be evaluated by appropriate approaches. We discuss them further in Section 4.

3 Metamodel and Tool

In the context of our work on quality in MDE, we have created a metamodel and supporting tool for the definition of quality models. The tool provides a graphical syntax that allows developers to define the important quality concepts of a given domain – in our example here, quality of models – and the relationships between these concepts. The metamodel can be seen a library of the most common artifacts that should be considered for any quality model, guiding the quality engineer in her/his task. Furthermore, once a quality model has been built, we can use transformations to generate useful artifacts from the model, like questionnaires, guidelines or checklists. In the following, we will provide an overview of our metamodel and tool.

3.1 The Metamodel

To support the ability of creating quality models for different targets (such as models or languages) and domains, we defined a metamodel. A part of this metamodel, representing the core concepts, is depicted in Fig. 3[3]. For a thorough explanation of the metamodel concepts we refer the reader to [2]. A brief summary follows.

[3] Missing are several subclasses of these core elements along with some detailing of relationships.

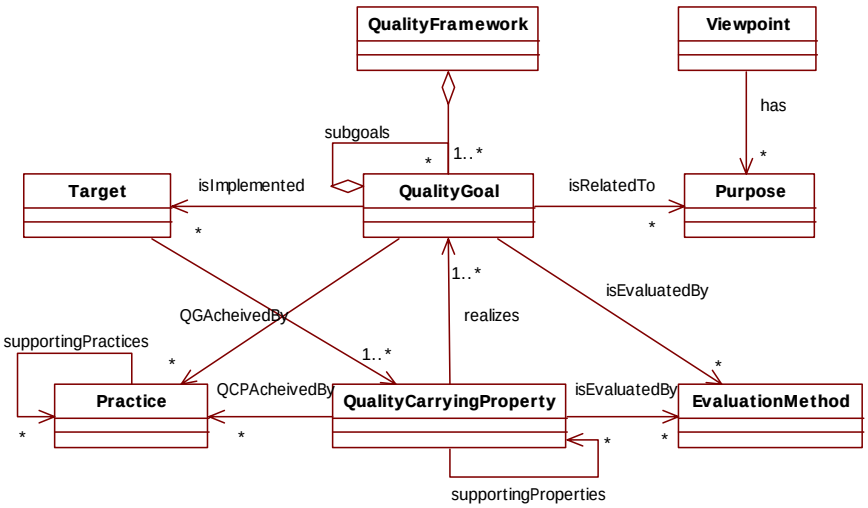


Fig. 3. Main constructs of the metamodel. Note that for improving the readability of the model we have omitted some elements as well as most of the aggregation relationships between QualityFramework and the other elements.

- A *QualityFramework* (or quality model) is a collection of quality entities and their relations. Quality frameworks can have *scope*. An example is whether a quality framework is generic or domain-specific.
- A *QualityGoal* is a clear and understandable definition of what quality means to a stakeholder such as users of a model, developers or managers. An example is to "improve software quality" of, for example, generated code. There may be a hierarchy of quality goals.
- A *QualityCarryingProperty* is some tangible property of an artifact or activity that is needed to achieve a quality goal. The purpose is to break down intangible quality goals into tangible properties of targets that can be evaluated. For example, understandability of models depends on them being simple and consistent. Quality-carrying properties may be supported by other properties.
- A *Target* is the artifact or activity that contains the property required to achieve a quality goal. For example, the quality of artifacts developed in an MDE approach depends on the quality of models, metamodels, tools, languages, transformations, modeling process and the expertise of people involved [1]. Therefore these elements are defined as target elements in the MDE quality framework.
- *Viewpoint* is used to indicate stakeholders of a quality goal such as system analyst, model developers or managers.
- *Purpose* describes what purpose a stakeholder, represented by a viewpoint, has in a given quality goal. For example, the purposes of modeling may be generation of code or documentation and these are related to specific quality goals.

- *Practice* is the means to achieve a required quality-carrying property. For example, using modeling conventions is a practice that can lead to developing correct and consistent models. Practices may be supported by other practices.
- Every property should be evaluated either quantitatively or qualitatively as defined in the *EvaluationMethod*. For example, including domain knowledge in a domain specific code generator (a practice in our model) will result in less error-prone code (a property) that can be evaluated by the reduction in the number of defects (a metrics).

All of the metamodel elements have the attributes name, definition (a textual description), type (in order to classify them if necessary) and evidence (connecting the element to literature).

3.2 Tool Support

The main point of the tool is, as mentioned, to support the use of the metamodel. Our goal is to allow people to define graphical quality models on different targets of a model-driven process, such as the tool, language, model, etc. We envision several benefits of our tool. Not only does such a quality model define what constitutes a high-quality target, but more importantly it systematically defines:

1. How high quality can be achieved during the development process, and
2. How to measure or evaluate whether or not this has been achieved.

Additionally, we plan to use such a model as the basis for:

1. Generating artifacts, like questionnaires, guidelines or checklists, by the use of model transformations.
2. Connecting evaluation methods, like metrics, model checking and simulation, to the quality model for evaluation purposes.

An early version of this tool has been implemented on the Eclipse platform using the Graphical Model Framework (GMF). Eclipse is widely used as a tool and development platform within academia and consequently provides several benefits; (1) people are experienced in using the environment, (2) using it promotes interoperability and allows our models to be used by other EMF-based tools, and (3) many plug-ins exist for possible reuse. The GMF plug-in, for example, allows one to create a concrete syntax in a graphical editor. Currently, our concrete syntax uses a UML profile-like syntax and consists of; a box, the concept name inside guillemots, a color and the name of the instanced concept itself. We also support showing the semantics of connectors by differentiating them either through graphics or tags.

This syntax is considered temporary, and our intention is to experiment with an increased use of graphics to differentiate concepts. The flexibility of GMF in defining graphics and icons is also a key reason for choosing GMF over UML-profiling for our tool solution. Additionally, a smaller metamodel is much better to use for model transformation and model validation purposes. Having an early implementation of the tool, our plan is to test it in a use case in order to gain experience with its usability and ability to model quality models. These experiences will also be the basis of the

following tool iterations, as well as implementing transformations and model evaluation techniques mentioned in the start of this sub-section.

4 An Instance of the Quality Model with Focus on the Quality of Models

We have performed a review of literature concerning approaches to improve the quality of models. This work is going to be submitted to a journal soon. Different publications have focused on different properties of models while our goal is to integrate these into a model that shows the C6 properties as discussed before and their relation to the practices proposed to improve the quality of models. This model is also used to examine the relations in the metamodel depicted in Figure 3. The target in our discussion in this paper is generally models.

There are three viewpoints when discussing the quality of models:

- Developers who need to understand models for the purpose of implementation and modification. For them, models should be defect-free in the first place and of course understandable;
- Tools that should interpret and analyze models or generate other artifacts from them. For them, models should be defect-free and technically comprehensible;
- Others who use models for the sake of communication. They need models that are understandable in the first place, and in less degree the defect-free aspect.

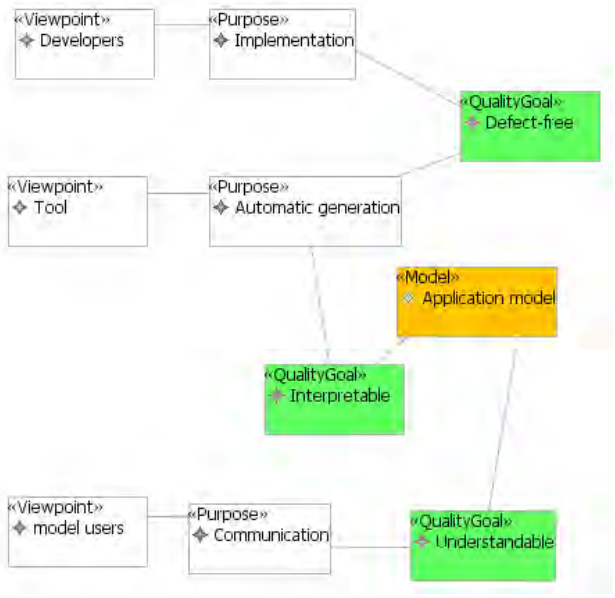


Fig. 4. Viewpoints, purposes of modeling and related quality goals

Thus we have defined three high-level quality goals: defect-free, interpretable by tools and understandable for human users (the last two are aspects of comprehensibility). Fig. 4 shows these quality goals. The quality model is developed by using the tool described in Section 3.2.

Being confined is also important if an organization has defined goals of modeling and one of its practices (using multiple views) might help understanding as discussed later. To be defect-free, a model should be correct, complete and consistent. To be understandable, a model should be appropriate relative to the domain, be well-organized, aesthetic and be close to the structure people have in mind of the problem domain. We have not shown comprehensibility as a property in the model since understandability and interpretability are defined as quality goals. Fig. 5 shows correctness and completeness properties important for a defect-free model. The relations between elements such as AcheivedBy and SupportingPractices are defined in the model while we have not assigned separate and specific graphics to them yet.

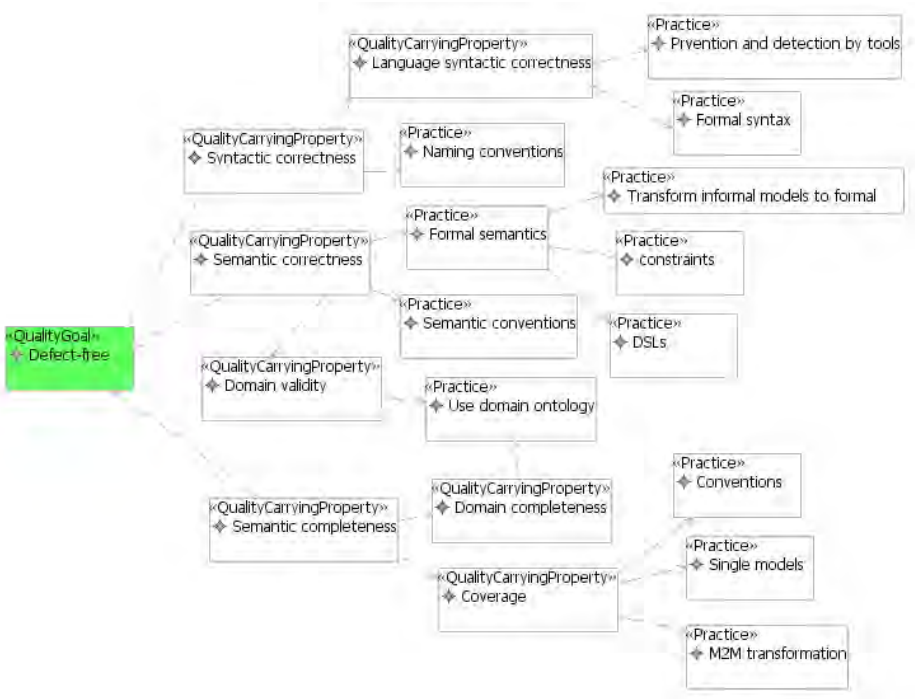


Fig. 5. A defect-free model should be correct (syntactically and semantically) and complete. The third property important is being consistent which is not shown in the figure.

The model also includes evaluation methods but these not included in the figure to improve readability. Language syntactic correctness and some semantic correctness and coverage may be evaluated by tools. Other properties such as domain validity and domain completeness are usually verified by inspecting the models.

Some of the practices we have identified to achieve quality properties are:

- Syntactic correctness may be improved if a modeling language has formal (or precise) syntax and by using different types of conventions such as naming conventions. Conventions are either provided as checklists or are enforced by tools.
- Semantic correctness: Domain validity (all the statements are relevant for the domain) may be improved by involving domain experts and using a domain ontology. Also using a DSL or UML profile, semantic conventions, semantic constraints, and transforming informal models into formal languages to allow analysis are practices to achieve semantic correctness.
- Completeness may be defined as domain completeness and coverage (for example all states are covered in the processes). We do not know of any practices to achieve domain completeness other that involving domain experts and using domain ontology. Coverage may be improved by having modeling conventions and generating models from other models (model-to-model transformations).
- Consistency may be improved by defining consistency constraints or conventions, performing Model-to-Model (M2M) transformation, and having a formal semantics.
- Understandability by human users or comprehensibility may be improved by well-organized models (using packaging conventions and possibility by limiting the number of diagrams), improving aesthetics (by layout conventions) and domain-appropriateness (by using concepts of the domain). Interpretability by tools will be improved by having a formal syntax and semantics.
- Confinement may be improved by having conventions on modeling, and clear definitions of the goals of modeling.

Finally, improving the modeling process impacts all of the quality goals by means of different practices such as defining goals of modeling.

Fig. 6 is a crosscut of the quality model related to understandability. The properties are best evaluated by inspections or experiments, while some layout issues may be evaluated by collecting metrics such as the number of classes in a class diagram.

Some practices help several quality properties. For example using stereotypes or DSLs (shown as DSLs in Fig. 5 and Fig. 6) allows adding formal semantics to models and helps domain-appropriateness and understandability, plus preventing errors in modeling and achieving semantic correctness. Other practices may help one property and have negative impact on another property. For example modeling a system from multiple views may help understanding but introduces consistency problems. Therefore we have defined relations such as “helps”, “breaks” and “depends” in our metamodel. However, showing all the details in the above figures would make them difficult to read.

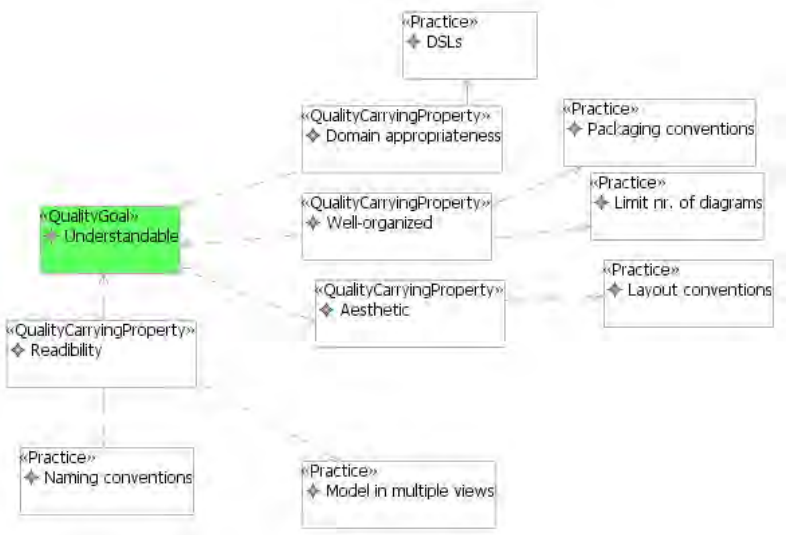


Fig. 6. Quality-carrying properties and practices proposed to improve the understandability of models

5 Discussion and Conclusion

This paper covers our attempt to identify quality properties of models and integrate earlier work on the quality of models. It also shows how to use a developed tool to build a quality model that allows us to specify and reason about the quality of models. Our quality model thus covers quality goals, properties to measure if the goals have been achieved and practices that help us in actually achieving them, in addition to evaluation methods. The most important properties for measuring the quality of models have been identified as correctness, completeness, consistency, comprehensibility and confinement.

The purpose of our work is two folded: Identifying concepts and practices related to the quality of models and other aspects important in model-drive engineering such as languages and transformations, and identifying general concepts that are needed to build quality models. Our metamodel serves the second purpose and defines a language that may be used when defining quality models. It defines the elements of quality models and their relations to one another and is built by integrating earlier work. We have so far built two quality models using this metamodel and tool related to the first goal; i.e., specifying quality goals in MDE. The first model with focus on the quality of domain-specific languages was presented in [2] while the second one with focus on the quality of models is presented in this paper. Building the metamodel and tool also allows us to experiment with concepts and relations and gain experience on the aspects we cover. Our metamodel and models should also be evaluated according to the quality goals we identify.

Future work will cover enhancing the tool and inserting literature results in various models. Having an early implementation of the tool, our plan is to test it in a use case in order to gain experience with its usability and ability to model an organization's quality goals and identify relevant properties and practices. These experiences will also be the basis of the following tool iterations.

Acknowledgments. This work has been funded by the Quality in Model-Driven Engineering project (cf. http://quality-mde.org/) at SINTEF.

References

1. Mohagheghi, P., Dehlen, V.: Developing a Quality Framework for Model-Driven Engineering. Models in Software Engineering, LNCS vol. 5002, pp. 275--286. Springer, Heidelberg (2008)
2. Mohagheghi, P., Dehlen, V.: A Metamodel for Specifying Quality Frameworks in Model-Driven Engineering. In: Proceedings of the Nordic Workshop on Model Driven Engineering, Engineering Research Institute, University of Iceland, pp. 51—65 (2008)
3. Dromey, R.G.: Concerning the Chimera. IEEE Software 13 (1), pp. 33--43 (1996)
4. Lindland, O.I., Sindre, G., Solvberg, A.: Understanding Quality in Conceptual Modeling. IEEE Software 11(2), pp. 42--49 (1994)
5. Krogstie, J.: Evaluating UML Using a Generic Quality Framework. Chapter in UML and the Unified Process, Idea Group Publishing, pp. 1--22 (2003)
6. Unhelkar, B.: Verification and Validation for Quality of UML 2.0 Models. Wiley-Interscience (2005)
7. Mitchell, R.: High-Quality Modeling in UML. In: The 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39), p. 388 (2001)
8. http://www.omg.org
9. Moody, D.L., Sindre, G., Brasethvik, T., Sølvberg, A.: Evaluating the Quality of Information Models: Empirical Testing of a Conceptual Model Quality Framework. In: The 25th International Conference on Software Engineering, pp. 295—305 (2003)
10. Haesen, R., Snoeck, M.: Implementing Consistency Management Techniques for Conceptual Modeling. In: 3rd International Workshop, Consistency Problems in UML-based Software Development III – Understanding and Usage of Dependency Relationships, pp. 99--113 (2004)
11. Nelson, H.J., Monarchi, D.E.: Ensuring the Quality of Conceptual Representations. Software Quality Journal 15(2), pp. 213--233 (2007)
12. Solheim, I., Neple, T.: Model Quality in the Context of Model-Driven Development. In: 2nd International Workshop on Model-Driven Enterprise Information Systems (MDEIS'06), pp. 27--35 (2006)
13. Harel D., Rumpe B.: Modeling Languages: Syntax, Semantics and All That Stuff. Technical paper number MCS00-16, The Weizmann Institute of Science, Rehovot, Israel (2000)